Formal methods and mechanical verification applied to the development of a convergent distributed sorting program.

T.E.J. Vos, S. D. Swierstra and **I.S.W.B. Prasetya**
Utrecht University, Department of Computer science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail: {tanja, doaitse}@cs.ruu.nl, wishnup@caplin.cs.ui.ac.id

September 30, 1996

Abstract

Gentle introductions to the programming logic UNITY, the theorem proving environment HOL, and the embedding of the first into the latter are presented. Equipped with this apparatus a methodology for designing distributed algorithms is described. Finally this methodology is used to design and proof the correctness of a convergent distributed sorting algorithm.

keywords: formal methods, UNITY, theorem provers, HOL, distributed algorithms, convergence.

Contents

1 Introduction 4

2 Why formal methods and mechanical verification 4

3 The results of a four-year study 5

4 Notational conventions in this report 6

5 HOL 7
5.1 Formulae in HOL 8
5.2 Theorems in HOL 9
5.3 Extending HOL 11
5.4 Proving theorems in HOL 12
5.5 Automatic theorem proving 14

6 UNITY **15**
6.1 States, predicates, expressions and actions 15
6.2 UNITY programs 18
6.2.1 The UNITY programming language 20
6.2.2 The well-formedness of a UNITY program 21
6.3 UNITY logic 22
6.4 Extended UNITY 26
6.4.1 Compositionality 26
6.4.2 Self-stabilisation 30

7 Embedding UNITY in HOL **34**
7.1 Predicates and Predicate Operators 35
7.2 Actions 35
7.3 UNITY programs 36
7.4 Program properties 37
7.4.1 Well-formedness 37
7.4.2 Safety, progress and self-stabilising properties 38

8 How to design and verify UNITY programs with HOL **40**

9 Case study **43**
9.1 Some notational conventions and necessary definitions 43
9.1.1 Notation 43
9.1.2 Relations and orderings 43
9.1.3 Graphs 43
9.1.4 Functions 45
9.2 Informally describe what the program is to do. 45
9.3 Create a formal specification of what the program is to do 46
9.4 Refine and decompose the specification 51
9.5 Construct a program which satisfies this refined specification 56
9.6 Represent the program in the HOL embedding of UNITY 63
9.7 Proof that the program is well-formed 64
9.8 Prove that the program satisfies the specification 70

10 Reflections **74**

Appendix **79**

A HOL definitions and pre-proved theorems **79**
A.1 Relations, orderings and permutations 79
A.2 Graphs and networks 80
A.3 Predicate Operators 80
A.4 Variables, Actions and Expressions 81
A.5 Core UNITY 81
A.6 Convergence 82
A.7 Parallel Composition 82

A.8 The sorting program and related definitions 83

References **85**

Index **88**

1 Introduction

In the last few years many computers have been connected to each other, and society has become more and more dependent on computer networks for their availability of world-wide communication. Distributed programs, underlying these world-wide communication networks, have to ensure that every message sent reaches its destination, and have to provide management for resources shared by various users on various computers. These are very complicated tasks. As a consequence of the increasing usage and dependence of computer networks, distributed programs have become of major importance. Accordingly, their trustworthiness must be subjected to severe tests of correctness.

In his dissertation Wishnu Prasetya [Pra95] reported the results of a four-year research-project on the mechanically supported verification of distributed algorithms. The objective of this present report is to summarise his findings and to show how these can be used to verify a distributed algorithm. The survey of Prasetya's results is given in section 3, which just recites them. Gentle introductions to the different aspects involved are given in sections 5 up to and including 7. The utilisation of these results is illustrated by means of a case study, which shows the formal design and mechanical verification of a convergent distributed program for sorting numbers according to an arbitrary total order.

2 Why formal methods and mechanical verification

Few if any would dispute that the use of computers within our society is not without risks. Nevertheless, computers are increasingly being used to monitor, and/or control, complex safety-critical processes where a run-time error or failure could result in death, injury, loss of property or environmental harm. Reasons for this growing employment of computers are that computers have the potential to increase power, improve performance, establish great efficiency, add flexibility and decrease costs. A consequence of this ever-increasing computerisation in safety-critical environments is that serious attention must be paid to the trustworthiness of the computer systems applied.

In his book [Neu95] Peter Neumann gives an extensive list of serious and less serious mishaps that have happened in computerised environments in the past few years. Deaths due to poorly designed software have occurred; for example the accidents that happened with the Therac-25, a computerised radiation therapy machine. Between June 1985 and January 1987, six known accidents involved massive overdoses by this machine – with resultant deaths and serious injuries. For a thorough investigation of these accidents with the Therac-25 the reader is referred to [LT93].

Formal methods, the term with which the variety of mathematical modelling techniques that are applicable to computer system design is meant, are often advocated as a way of increasing confidence in computer based systems. Many [BS92, BH95b, BS93b, BBL93, BH95a, BS93a, Bow93, But93, CGR93, CG92, GCR94, Hal90, Kem90, Nic91, RvH93, Rus94, WW93] believe that the use of formal methods currently offers the only intellectually defensible method for handling the software crisis which increasingly affects the world of embedded systems. In this report we shall mainly concentrate on safety-critical software design. Formal methods can be applied at three levels, which provide different levels of assurance for the software developed.

At a basic level, formal methods may be used for specification of the system to be designed. The use of formal specification techniques can be of benefit in most cases. Using a formal specification language instead of natural language has the advantages that specifications are more concise and less ambiguous, which makes it easier to reason about them and helps to gain greater insight into and understanding of the problem solved. Furthermore, formal specifications serve as a valuable piece of documentation, which is essential for software maintenance purposes. The next level of use is formal development, which involves formally specifying the program, proving that certain properties are satisfied, proving that undesirable properties are absent, and finally applying a refinement and decomposition calculus to the specification such that it may gradually be translated into an efficient and concrete representation of the program. The proofs involved are pencil-and-paper proofs, which can be formal or informal, depending on the level of assurance that is required. At the last, and most rigorous, level, the whole process of proof is mechanised. Hand proofs or design inevitably lead to human error occurring for all and even the simplest systems. Verifying the design process with a mechanical theorem prover reduces the possibility of errors. Although some argue that this can never eliminate errors completely since the program that does the verification itself may be incorrect, experience shows that theorem provers are very reliable, and definitely least much more reliable than people. In addition to reducing errors, the use of theorem provers attributes to the understanding of the problem that is being solved, because during the verification process one is irrevocable confronted with every aspect of the program under construction.

In using one of these levels, one has to determine which level is suitable for the problem at hand. Before deciding to use full formal development and mechanised proofs, one must resolve whether the additional costs (in time, effort, manpower, tool support, money, education) is justified. For safety critical systems – that is computer based systems, in which a system failure could result in death, injury, environmental harm or loss of property, money or information – such additional costs are warranted and required.

Although, formal methods and automatic verification are becoming more and more accepted as the only intellectually defensible way in which the quality of both software and hardware can be improved, it should, however, be remembered that they are *not* some universal panacea. To quote from C.A.R. Hoare:

> Of course, there is no fool-proof methodology or magic formula that will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgement, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalise and communicate design decisions, and help to ensure that they are correctly carried out.

3 The results of a four-year study

First, an embedding of the programming logic UNITY was made inside the theorem prover HOL, by extending the latter with all definitions required by the logic, and making all basic theorems of the logic available by proving them. The gain from this is that the formal design of programs can now be assisted by a mechanical verification with the theorem prover, resulting in a significant increase of the trustworthiness of the design.

Second, two extensions of the programming logic UNITY were defined and embedded within the theorem prover HOL. The first extension of the programming logic UNITY concerns compositionality properties. A problem of UNITY is that progress properties are not compositional with respect to parallel composition. That is, we cannot in general factorise a progress specification of a program into the specifications of its parallel components. Therefore one is unable to develop a component program in isolation, which is awkward. The extension presented is however compositional. The second extension regards self-stabilisation and convergence of programs. Roughly speaking, a self-stabilising program is a program which is capable of recovering from arbitrary transient failures. Obviously such a property is very useful, although the requirement to allow arbitrary failures may be too strong. The more general notion of convergence, which allows a program to recover only from certain failures, is used to express a more restricted form of self-stabilisation. Since self-stabilisation and convergence are considered to be essential for programs in safety-critical environments, e.g. distributed environments, this second extension is significant for our purposes. Moreover, an induction principle is formulated for convergence which is stronger than the one for UNITY's reach-to operator. As a consequence, a powerful technique for proving convergence has become available.

Third, this HOL embedding of extended UNITY was used to mechanically verify Lentfert's Fair and Successive Approximation (FSA) algorithm [Len93]. This algorithm is a complicated self-stabilising distributed program which computes the minimal distance between all pairs of vertices in a network, in other words an algorithm which maintains routing tables for a network in a changing environment (that is, vertices and links may appear and disappear during the execution of the algorithm).

Last, two generalisations of Lentfert's FSA algorithm were defined and proved with HOL. The first generalisation extends the applicability of the FSA algorithm, by generalising the notion of 'minimal distance'. This generalised FSA algorithm computes a certain class of minimal-distance-like functions, called round-solvable functions. In order to prove this generalisation, large HOL-libraries have been developed for orderings and round-solvability of functions. The second generalisation lifts the FSA algorithm so that it works for hierarchical networks of 'domains' instead of ordinary networks of vertices.

4 Notational conventions in this report

When computer checked results (definitions) are presented, they shall be marked by the names they are identified with in the HOL theories (definitions) that we wrote. For example:

Theorem 4.1 PINK PANTHER *Pink_Panther_thm*

$$Fu \;=\; Fu \circ Fu$$

The number and the name PINK PANTHER are how we refer to the theorem in this thesis. The name `Pink_Panther_thm` is how the theorem is called in HOL. *Implicitly, this means that the theorem has been mechanically verified.*

When presenting a theorem we often present it like this:

$$\begin{array}{c} A_1 \\ A_2 \\ \hline B \end{array}$$

which is another way for denoting $A_1 \wedge A_2 \Rightarrow B$. The notation is commonly used within the UNITY community.

5 HOL

For those who are familiar with HOL, this section can be skipped.

The HOL system [GM93] is one of the most widely used theorem provers, both in academia and industry. It is free, comes with extensive documentation, libraries, an interactive help system, and myriad web-sites providing information and a dynamic search engine for HOL information[1]. HOL is a direct descendant of the innovative LCF (Logic of Computable Functions) [Pau87] theorem prover developed by Robin Milner in the early 1970s, and is an implementation of a version of Church's simple theory of types, a formalism dating back more than 50 years. HOL is an acronym of Higher Order Logic, the logic used by the HOL system. Basically, this logic is first-order classical predicate logic, with the following differences: the logic is higher-order (variables can range over functions and predicates); the logic is typed; and there is no separate syntactic category of formulae (terms of boolean type play the role of formulae).

HOL is an interactive theorem prover: one types a formula and proves it step by step using any primitive strategy provided by HOL. When the proof is completed, the code can be collected and stored in a file, to be given to others for the purpose of re-generating the proved fact, or simply for documentation purposes in case modifications are required.

HOL is built on top of the strict functional programming language ML. The HOL system defines ML types for the various logical entities (e.g. terms, types, theorems, theories). The ML type for theorems (denoted in ML as `thm`) is an abstract data-type, with operators corresponding to the axioms and inference-rules. In this way ML's type system ensures that only valid inferences can be made. One of the main strengths of HOL is the availability of ML as a meta-language. In HOL, ML is used to program proof *tactics*. The notion of tactics was invented in 1970 by Robin Milner. In HOL a tactic is an ML function with which one can do goal-directed (i.e. top-down) proofs by splitting a theorem (the top-goal) into a number of simpler parts (the subgoals), proving these separately, and then combining the proofs of the subgoals to construct the proof of the whole.

HOL is not generally attributed to be an automatic theorem prover. Full automation is only possible if the scope of the problems at hand is limited. Instead HOL provides a general platform which, if necessary, can be fine-tuned to the specific kind of proofs to be constructed.

The major hurdle in using HOL is that it is, after all, still a machine which needs to be told in detail what to do. When a formula needs to be re-written in a subtle way, for humans this may seem one of the simplest things that there is. For a machine, however, it is not so simple since it needs to know precisely which variables have to be replaced, the positions at which they are to be replaced, and by what they should be replaced. On the one hand HOL has a whole range of tools to manipulate formulae: some designed for global operations such as replacing all x in a formula with y, and some for finer surgical operations such as replacing an x at a particular position in a formula with something else. On the other hand it does take quite a while before one gets a sufficient grip on what exactly each

[1] http://lal.cs.byu.edu/lal/hol-documentation.html

	standard notation	HOL notation
Denoting types	$x \in A$ or $x : A$	"x:A"
Proposition logic	$\neg p$, true, false	"~p", "T", "F"
	$p \wedge q$, $p \vee q$	"p /\ q", "p \/ q"
	$p \Rightarrow q$	"p ==> q"
Universal quantification	$(\forall x, y :: P)$	"(!x y. P)"
	$(\forall x : P : Q)$	"(!y::P. Q)"
Existential quantification	$(\exists x, y :: P)$	"(?x y. P)"
	$(\exists x : P : Q)$	"(?x::P. Q)"
Function application	$f.x$	"f x"
λ abstraction	$(\lambda x.\ E)$	"(\x. E)"
Conditional expression	if b then E_1 else E_2	"b => E1 | E2"
Sets	$\{a, b\}$, $\{f.x \mid P.x\}$	"{a,b}", "{f x | P x}"
Set operators	$x \in V$, $U \subseteq V$	"x IN V", "U SUBSET V"
	$U \cup V$, $U \cap V$	"U UNION V", "U INTER V"
	$U \backslash V$	"U DIFF V"
Lists	$a;s$, $s;a$	"CONS a s", "SNOC a s"
	$[a; b; c]$, st	"[a;b;c]", "APPEND s t"

Figure 1: The HOL Notation.

tool does, and how to use them effectively. Perhaps, this is one thing that scares some potential users away.

Another problem is the collection of pre-proven facts. Although HOL is probably a theorem prover with the richest collection of facts, compared to the knowledge of a human expert, it is a novice. A simple fact such as $(\forall a, b :: (\exists x :: ax + b \leq x^2))$ may be beyond HOL's knowledge. When a fact is unknown, users will have to prove it themselves. Many users complain that their work is slowed down by the necessity to "teach" HOL sundry simple mathematical facts. At the moment, various people are working on improving and enriching the HOL library of facts.

5.1 Formulae in HOL

All HOL formulae, also called terms of the HOL logic or logical terms, are represented in ML by a type called `term`. Figure 1 shows examples of how the standard notation is translated to the HOL notation. As the reader can see, the HOL notation is as close an ASCII notation can be to the standard notation. Anything between a pair of quotes is parsed as a logical term. Terms of the HOL logic are quite similar to ML expressions, which can cause confusion since these terms have two types, logical types and ML types (called *object language types* and *meta-language types* respectively). The object language types of HOL terms are also represented by an ML type called type. These object language types are denoted by expressions of the form `": ... "`. There is a built-in function type_of, which has ML type `term->type` and returns the logical type (i.e object language type) of a HOL term. An example taken from [GM93] may elucidate these matters. Consider for example the logical term `"(1,2)"`

- It is a HOL term with object language type (i.e. logical type) `":num#num"`.
- It is an ML expression with meta-language type `term`
- Evaluating `type_of "(1,2)"` results in `":num#num"`. This object language type `":num#num"` has ML type `type`.
- In contrast consider the ML expression `("1","2")` which has ML type `term#term`.

Within a quotation (i.e. `" .... "`), expressions of the form `^(t)` (where `t` is an ML expression of meta-language type `term` or `type`) are called *anti-quotations.* An anti-quotation `^(t)` evaluates to the ML value of the expression `t`. As an exemplification, consider the following small HOL-session:

```
#let x = "y /\ z";;
x = "y /\ z" : term

#let p = "^x \/ k";;
p = "y /\ z \/ k" : term

#
```

There are different kinds of object language types. First, there are *primitive types* such as `":bool"`, `":nat"` and `":ind"` (representing the Boolean set of `true` and `false`, the natural numbers, and the infinite set respectively). Second, there are *type-variables* to denote, as the name suggests, arbitrary types. Names denoting type-variables must always be preceded by a `*` like in `":*A"` or `":*B"`. Third, object language types can be combined using type constructors (e.g. product (`#`), function (`->`), lists (`list`), sets (`set`)) to form new *compound types.* In HOL one can define new type constructors with the ML function `define_type`. For example the type operator `triple` can be defined as follows:

```
define_type `triple_DEF` `triple = TRIPLE *A *B *C`
```

After the execution of this function `"x:(num,bool,bool)triple"`, for example, is a valid HOL-term.

5.2 Theorems in HOL

A **theorem** is an ML expression of meta-language type `thm`. Theorems can only be created by a formal proof. A formal proof is a sequence, each element of which is either an axiom or can be derived from earlier elements of the sequence by a rule of inference. Consequently, each element of the sequence is a theorem proved by the formal proof. HOL's deductive system has only eight primitive inference rules and five axioms, everything else is formally derived. And since creating theorems can only[2] be done by constructing a formal proof, this makes for a reliable proof system.

More specifically, a HOL theorem has the form:

[2]Because `thm` is an abstract data type with operators corresponding to the axioms and inference rules.

```
A1; A1; ... |- C
```

where the `Ai`'s are boolean HOL terms representing assumptions and `C` is boolean HOL term representing the conclusion of the theorem. It is to be interpreted as: if all `Ai`'s are valid, then so is `C`. An example of a theorem is the following:

```
|- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)
```

which is the induction theorem on natural numbers[3].

The first four axioms in HOL are:

```
BOOL_CASES_AX   :  |- !t. (t=T) \/ (t=F)
IMP_ANTISYM_AX :  |- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1) ==> (t1=t2)
ETA_AX          :  |- !t. (\x. t x) = t
SELECT_AX       :  |- !(P:*->bool) x. P x ==> P ($@ P)
```

where `@` denotes the choice operator. `$@ P` picks an element x such that it satisfies `P` —if such an element exists (which is exactly what the fourth axiom above says). As for the other axioms: the first states that a boolean term is either true or false[4]; the second axiom states that $\Rightarrow$ is anti-symmetric; and the third states that $(\lambda x.f\ x) = f$.

The fifth axiom asserts that there exists a injection between the type `ind` and itself, but the injection is not surjective. In other words, the type `ind` is infinite. The axiom is given below:

```
INFINITY_AX     :  |- ?f:ind->ind. ONE_ONE f /\ ~(ONTO f)
```

The 8 primitives inference rules are:

i. `ASSUME`, introduces an assumption: `ASSUME "t"` generates the theorem `t |- t`.

ii. `REFL`, yields a theorem stating that any HOL-term `t` is equal to itself: `REFL "t"` generates `|- t=t`.

iii. `BETA_CONV`, corresponds to Beta reduction. For example, `BETA_CONV "(\x. (x + 1)) X"` generates `|- (\x. (x+1)) X = (X+1)`.

iv. `SUBST`, is used to perform a substitution within a theorem.

v. `ABS`, generates a theorem, e.g. `ABS "t1 = t2"` generates the theorem `|- (\x. t1) = (\x. t2)`.

vi. `INST_TYPE`, is used to instantiate type variables in a theorem.

vii. `DISCH`, is used to discharge an assumption. For example, `DISCH (A1; A2 |- t)` yields `A1 |- A2 ==> t`.

viii. `MP`, applies the Modus Ponens principle. For example, `MP (|- t1 ==> t2) (|- t1)` generates `|- t2`.

More sophisticated inference rules can be derived by combining the above primitive rules.

Examples of (frequently used) derived rules are `REWRITE_RULE` and `MATCH_MP`. Given a list of equational theorems, `REWRITE_RULE` tries to rewrite a theorem using the supplied

[3]All variables which occur free are assumed to be either constants or universally quantified.
[4]Hence the HOL logic is conservative.

equations. The result is a new theorem. `MATCH_MP` implements the modus ponens principle. In illustration below are some HOL sessions.

```
1 #DE_MORGAN_THM ;;
2 |- !t1 t2. (~(t1 /\ t2) = ~t1 \/ ~t2) /\ (~(t1 \/ t2) = ~t1 /\ ~t2)
3
4 #th1 ;;
5 |- ~(p /\ q) \/ q
6
7 #REWRITE_RULE [DE_MORGAN_THM] th1 ;;
8 |- (~p \/ ~q) \/ q
```

The line numbers have been added for our convenience. The `#` is the HOL prompt. Every command is closed by `;;`, after which HOL will return the result. On line 1 HOL is asked for the value of `DE_MORGAN_THM`. HOL returns on line 2 a theorem, de Morgan's theorem. Line 4 shows a similar query. On line 7 HOL is requested to rewrite theorem `th1` with theorem `DE_MORGAN_THM`. The result is found on line 8.

The example below shows an application of the modus ponens principle using the `MATCH_MP` rule.

```
1 #LESS_ADD ;;
2 |- !m n. n < m ==> (?p. p + n = m)
3
4 #th2 ;;
5 |- 2 < 3
6
7 #MATCH_MP LESS_ADD th2;;
8 |- ?p. p + 2 = 3
```

5.3 Extending HOL

The core of HOL provides predicate calculus. To use it for a particular purpose, it has to be extended. For instance, to verify programs with HOL, the latter must be told what programs and specifications are. Two ways exist to extend HOL: by adding axioms or by adding definitions. Adding axioms is considered dangerous because inconsistency can easily arise when axioms are introduced freely. Although it is also possible to introduce absurd definitions, these are nothing more than abbreviations, and hence cannot introduce inconsistency. Definitional extension, also called *conservative extension*, is therefore a much more preferred practice.

In HOL a **definition** is also a theorem, which states the meaning of the object that is being defined. Because the HOL notation is quite close to the standard mathematical notation,definition of new objects can, to some extend, be written in a natural way. Below Hoare triples are defined.

```
#let HOA_DEF = new_definition
  (`HOA_DEF`,
   "HOA (p,a,q) =
      (!(s:*) (t:*). p s /\ a s t ==> q t)") ;;

HOA_DEF = |- !p a q. HOA(p,a,q) = (!s t. p s /\ a s t ==> q t)
```

◀

5.4 Proving theorems in HOL

To prove a conjecture one can start with known facts, combine these to deduce new facts, and continue until the conjecture is obtained. Alternatively, one can start with the conjecture, work backwards by splitting it into new conjectures, and continue until all obtained conjectures can be reduced to known facts. The first yields what is called a *forward proof* and the second yields a *backward proof*. This can be illustrated by the tree in Figure 2. It is called a proof tree. At the root of the tree is the conjecture. The tree is said to be closed if all leaves are known facts, and hence the conjecture is proven if a closed proof tree is constructed. A forward proof attempts to construct such a tree from bottom to top, and a backward proof from top to bottom.

In HOL, new facts can only be generated by applying HOL inference rules to known facts, i.e. axioms and previously proved facts; basically this comprises forward proof in HOL. There is, however, also support for the construction of backward proofs. A conjecture is called a *goal* in HOL. It has the same structure as a theorem:

```
A1; A2; ... ?- C
```

Note that a goal is denoted with `?-`, whereas in a theorem we write `|-`. To manipulate goals, *tactics* are available. A tactic may prove a goal —that is, convert it into a theorem. For example `ACCEPT_TAC` proves a goal `?- g` if `g` is a known fact. A tactic may also transform a goal into new goals —or *subgoals*, as they are called in HOL—, which hopefully are easier to prove.

Examples of other tactics are: rewrite tactics, which rewrite (or solve) a goal by using as rewrite rules (i.e. as left-to-right replacement rules) the conclusions of a given list of equational theorems; decomposition tactics, for example `CONJ_TAC`, which splits a conjunctive goal `?- g1 /\ g2` into two separate subgoals `?- g1` and `?- g2`; matching tactics, e.g. `MATCH_MP_TAC`, which when applied to a goal `?- g` uses a supplied implication (e.g. `|- p ==> g`) to reduce the goal to the subgoal `?- p`; resolution tactics, for example `RES_TAC`, which tries to generate more assumptions by applying, among other things, modus ponens to all matching combinations of the assumptions. For instance, applying `RES_TAC` to the goal:

```
"0<x"; "!y. 0<y ==> z<y+z"; "z<x+z ==> g"   ?-   "g"
```

will yield the following new goal:

```
"z<x+z"; "0<x"; "!y. 0<y ==> z<y+z"; "z<x+z ==> g"   ?-   "g"
```

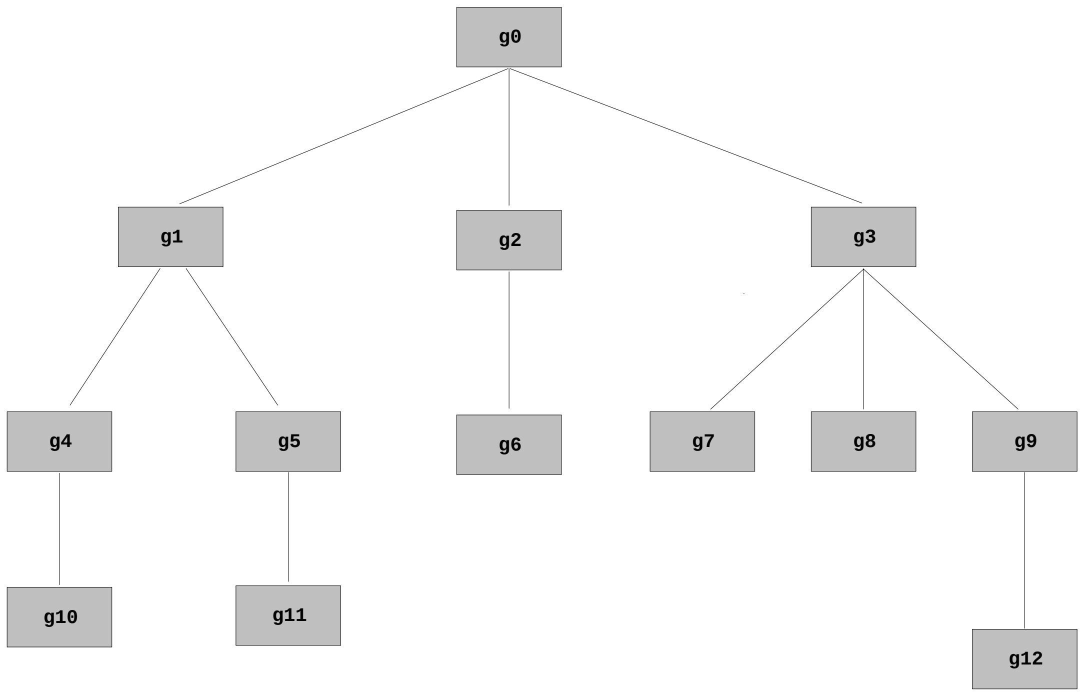

Figure 2: A proof tree.

Applying `RES_TAC` again to the new goal will generate "g", the tactic then concludes that the goal is proven, and returns the corresponding theorem.

Tactics are not primitives in HOL. They are built from inference rules. When applied to a goal `?- g`, a tactic does not only generate new goals —say, `?- g1` and `?- g2`— but also a justification function. Such a function is a rule, which if applied, in this case, to theorems of the form `|- g1` and `|- g2` will produce `|- g`. When a composition of tactics proves a goal, what it basically does is re-building the corresponding proof tree from the bottom, i.e. the known facts, to the top, i.e. the goal, using the generated justification functions to construct new nodes up in the tree.

HOL provides much better support for backward proofs than it does for forward proofs. For backward proofs, HOL provides higher order tactics, also called tactics combinators or *tacticals*, with which tactics can be composed. Examples of the most used tacticals in HOL are `THEN`, `ORELSE` and `REPEAT`.

- `THEN : tactic -> tactic -> tactic`; if `T1` and `T2` are tactics, then `T1 THEN T2` is a tactic, which first applies `T1` and then applies `T2` to all the subgoals that were generated by `T1`.
- `ORELSE: tactic -> tactic -> tactic`; if `T1` and `T2` are tactics, then `T1 ORELSE T2` is a tactic, which first tries `T1`. If `T1` fails then it tries `T2`.

- `REPEAT: tactic -> tactic`; if `T` is a tactic, then `REPEAT T` is a tactic that repeatedly applies `T` until it fails.

On the other hand, no rules combinators are provided. Although, using the meta language ML rules combinators can quite easily be made.

HOL also provides a facility, called the *sub-goal package*, to interactively construct a backward proof. The package will memorise the proof tree and justification functions generated in a proof session. The tree can be displayed, extended, or partly un-done. Whereas interactive forward proofs are also possible in HOL simply by applying rules interactively, HOL provides no facility to automatically record proof histories (proof trees).

5.5 Automatic theorem proving

As higher order logic —the logic that underlies HOL— is not decidable, there exists no decision procedure that can automatically decide the validity of each HOL formula. However, for limited applications, it is often possible to provide automatic procedures. The standard HOL package is supplied with a library called `arith` written by Boulton [Bou94]. The library contains a decision procedure to decide the validity of a certain subset of arithmetic formulae over natural numbers. The procedure is based on the Presburger natural number arithmetic [Coo72]. An exemplification is:

```
1 #set_goal([],"x<(y+z) ==> (y+x) < (z+(2*y))") ;;
2 "x < (y + z) ==> (y + x) < (z + (2 * y))"
3
4 #expand (CONV_TAC ARITH_CONV) ;;
5 goal proved
6 |- x < (y + z) ==> (y + x) < (z + (2 * y))
```

◀

To prove $x < y + z \Rightarrow y + x < z + 2y$: first the goal is set on line 1, then the Presburger procedure, `ARITH_CONV`, is invoked on line 4, which immediately proves the goal.

There is also a library called `taut` to check the validity of a formula from propositional logic. This library can for instance be used to automatically prove $p \wedge q \Rightarrow \neg r \vee s = p \wedge q \wedge r \Rightarrow s$. It cannot be used to prove more sophisticated formulae from predicate logic, such as $(\forall x :: P.x) \Rightarrow (\exists x :: P.x)$ (assuming a non-empty domain of quantification). There is, however, a library called `faust` written by Schneider, Kropf, and Kumar [SKR91] that provides a decision procedure to check the validity of many formulae from first order predicate logic. The procedure can handle formulae such as $(\forall x :: P.x) \Rightarrow (\exists x :: P.x)$, but not $(\forall P :: (\forall x : x < y : P.x) \Rightarrow P.y)$ because the quantification over P is a second order quantification (no quantification over functions is allowed).

So some automatic theorem proving capacity is available in HOL, but it is limited. The `arith` library cannot, for example, handle multiplication[5]. Temporal properties of a program, such as we are dealing with in UNITY, are often expressed in higher order formulas, and hence cannot be handled by `faust`.

[5]In general, natural number arithmetic is not decidable if multiplication is included.So the best we can achieve is a partial decision procedure.

6 UNITY

Basically, a program is a collection of actions. During the execution of a program, its actions are executed in a certain order. In considering parallel or distributed executions of programs, strict orderings on when, where and how actions are executed vanish. Therefore, many distributed programming logics view programs as a collections of actions without any ordering. UNITY [CM88] consists of such a programming theory. It is invented to support the design and verification of distributed programs. An important aspect of UNITY is the separation of concerns between the core problem to be solved by an algorithm and the implementation details concerning programming languages and architectures. The UNITY theory consists of a programming language and an associated logic. Program design in UNITY commences with the formulation of a program specification in the UNITY logic. Next, this specification is refined until it is detailed enough to be directly associated with a UNITY program. A UNITY program describes *what* must be done, in other words it describes the initial state and the state transitions. A UNITY program does not describe *when*, *where* and *how* actions are executed. As a result, detailed architectural concerns do not have to be considered during the design of a UNITY program.

6.1 States, predicates, expressions and actions

First, a brief review on some basic notions in programming is given. A program has a set of variables. The values of these variables at a given moment is called the *state* of the program at that moment. Assume a universe Var of *all* program variables, and a universe Val of all values these variables may take. Let P be a program and V be a set consisting of all variables of P, that is $V \subseteq \mathsf{Var}$. A state of P can be represented by a function $s \in V \rightarrow \mathsf{Val}$. The value of a variable v in state s is denoted by $s.v$. However, since in HOL all functions are required to be total and since sub-typing is not possible, we shall represent a state of P by a *total* function $s \in \mathsf{Var} \rightarrow \mathsf{Val}$. The value of a variable v outside V in a state s is irrelevant to P in the sense that it cannot influence any execution of P and neither can any execution of P influence it. The set of all (program) states shall be denoted by State. For a state $s \in \mathsf{Var} \rightarrow \mathsf{Val}$, the projection or restriction of s to a set V, denoted by $s \upharpoonright V$ can be defined as a *partial* function of type $V \rightarrow \mathsf{Val}$ such that $(s \upharpoonright V).x = s.x$ if $x \in V \cap \mathsf{Var}$ and else it is undefined. Since, the projection of a state must be a state again (i.e. a total function), a constant $\aleph$ is introduced. In HOL, all objects are typed. A type defines of course a set. The types in HOL are all non-empty. All that is known about $\aleph$ is that for any given (non-empty) type T, $\aleph_T$ exists and is a member of T. The constant $\aleph$ can be seen as 'undefinedness' in a partial function. However, evaluating an 'undefined' value results in an error and as a consequence special calculation rules have to be added to handle these errors. Therefore we prefer to regard $\aleph$ as an ordinary value representing the set of 'uninteresting' but otherwise valid values. Hereby the need to introduce special rules to handle 'undefinedness' is eliminated.

The following definition formally defines projection of an arbitrary function:

Notation	Meaning	HOL-definition
true	$(\lambda s.\ \mathsf{true})$	TT_DEF
false	$(\lambda s.\ \mathsf{false})$	FF_DEF
$\neg p$	$(\lambda s.\ \neg p)$	pNOT_DEF
$p \Rightarrow q$	$(\lambda s.\ p.s \Rightarrow q.s)$	pIMP_DEF
$p \wedge q$	$(\lambda s.\ p.s \wedge q.s)$	pAND_DEF
$p \vee q$	$(\lambda s.\ p.s \vee q.s)$	pAND_DEF
$(\forall i : W.i : P.i)$	$(\lambda s.\ (\forall i : W.i : P.i.s))$	RES_qAND
$(\exists i : W.i : P.i)$	$(\lambda s.\ (\exists i : W.i : P.i.s))$	RES_qOR

Note: p and q are predicates over State. The dummy s ranges therefore over State.

Table 1: Overloading of the boolean operators.

Definition 6.1 PROJECTION *Pj_DEF*
For all $f \in A \to B$, $V \subseteq A$, and $x \in A$:

$$(x \in V \ \Rightarrow\ (f \upharpoonright V).x = f.x)\ \wedge\ (x \notin V \ \Rightarrow\ (f \upharpoonright V).x = \aleph)$$

A predicate over a set S is a function of type $S \to \mathbb{B}$. In particular, we are interested in predicates over State, since these are used to specify pre and post conditions of a program. Such predicates are called a *state-predicates*. A state-predicate is used to describe a set of states satisfying a certain property. For instance:

$$(\lambda s.\ (s.x = s.y + 1)\ \wedge\ n < s.y)$$

is a state-predicate describing all states s in which the value of variable x is the value of variable y plus 1 and the value of variable y is greater than some constant n. In practice, the above state-predicate is usually written as:

$$(x = y + 1)\ \wedge\ n < y$$

So, the symbols $=$, $+$, $\wedge$ and $<$ are being overloaded. Other examples of such overloading are:

"$y < x$", "$p \wedge q$", or "$(\exists i : P.i : x.i = 0)$"

which actually denote:

"$(\lambda s.\ s.y < s.x)$", "$(\lambda s.\ p.s \wedge q.s)$", and "$(\lambda s.\ (\exists i : P.i : s.(x.i) = 0))$"

Usually, this kind of overloading does not cause confusion. There are, however, occasions where a careful distinction is called for. Therefore, it is important that the reader is well aware of this double interpretation. To emphasise this, Table 1 shows the "lifted" meaning of the standard boolean operators. The set of all state-predicates shall be denoted with Pred.

A predicate p over S is said to hold *everywhere* —usually denoted by $[p]$— if $p.s$ holds for all $s \in S$.

A state-predicate p is said to be *confined* by a set of variables V, denoted by $p \in \mathsf{Pred}.V$, if p does not restrict the value of any variable outside V (unless p is already empty):

Definition 6.2 PREDICATE CONFINEMENT *CONF_DEF*

$$p \in \mathsf{Pred}.V \;=\; (\forall s,t :: (s{\upharpoonright}V = t{\upharpoonright}V) \Rightarrow (p.s = p.t))$$

◀

For example, $x+1<y$ is confined by $\{x,y\}$ but not by $\{x\}$. Notice that if p is confined by V, p does not contain useful information about variables outside V. Indeed, $p \in \mathsf{Pred}.V$ is how we encode $p \in (V {\rightarrow} \mathsf{Val}) {\rightarrow} \mathbb{B}$.

State-predicates true and false are confined by any set. Confinement is preserved by any predicate operator in Table 1. So, for example, if $p,q \in \mathsf{Pred}.V$ then $p \wedge q \in \mathsf{Pred}.V$. As a rule of thumb, any predicate p is confined by $\mathsf{free}.p$, that is, the set of variables occurring free in p:

$$p \in \mathsf{Pred}.(\mathsf{free}.p) \tag{6.1}$$

Note however, that $\mathsf{free}.p$ is not necessarily the smallest set which confines p. For example, $\emptyset$ confines "$0 = x \vee 0 \neq x$".

Analogue to predicates over State, we call expressions over State (i.e. functions of type $\mathsf{State}{\rightarrow}\mathsf{Val}$) *state-expressions.* As we already saw above these expressions are also being overloaded, that is "$y+1$" is written in stead of "$(\lambda s.s.y+1)$". Furthermore, "e_1+e_2" is the overloaded notation for "$(\lambda s.e_1.s + e_2.s)$". Note that state-predicates are actually a special kind of state-expressions, viz. those where Val is $\mathbb{B}$.

An action (statement) of a program can change the state of a program. An obvious way to represent an action is by a function a where the state resulting from the execution of a on a state s is given by $a.s$. As a result, such an action is always deterministic. To allow for non-determinism, we will represent an action as a relation on State. That is, an action a has the type $\mathsf{State}{\rightarrow}\mathsf{State}{\rightarrow}\mathbb{B}$. The interpretation of $a.s.t$ is that t is a possible state resulting from execution of a at state s. Consequently, Hoare triple can be defined as follows:

$$\{p\}\ a\ \{q\} \;=\; (\forall s,t :: p.s \wedge a.s.t \Rightarrow q.t) \tag{6.2}$$

All kind of basic laws for Hoare triples are derivable from this definition. For example:

$$\frac{(\{p\}\ a\ \{q\}) \wedge (\{r\}\ a\ \{s\})}{\{p \wedge r\}\ a\ \{q \wedge s\}} \quad \text{and} \quad \frac{[p \Rightarrow q] \wedge (\{q\}\ a\ \{r\}) \wedge [r \Rightarrow s]}{\{p\}\ a\ \{s\}}$$

The set of all actions will be called Action, examples of elements in this set are given below, where for some set S, S^{c} denotes the complement of set S (i.e. $\{x\}^{\mathsf{c}}$ is the set $\mathsf{Var} - \{x\}$).Action (6.3) is an assignment action, which assigns the value of state-expression e to the variable x and, except for x, does not change the value of any other variable. The guarded action (6.4) executes action a or b, dependent on whether guard g is true or not. The skip-action (6.5) does not change any variable.

$$\begin{aligned} \mathsf{assign}.x.e &= (\lambda s,t.\ (t.x = e.s) \wedge (t\restriction\{x\}^{\mathsf{c}} = s\restriction\{x\}^{\mathsf{c}})) && (6.3)\\ \mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b &= (\lambda s,t.\ (g.s \Rightarrow a.s.t) \wedge (\neg g.s \Rightarrow b.s.t)) && (6.4)\\ \mathsf{skip} &= (\lambda s,t.\ s = t) && (6.5) \end{aligned}$$

Projection $\restriction$ of functions can be lifted to the action level as follows:

Definition 6.3 $\restriction$ ON Action *a_Pj_DEF*

$$(a\restriction V).s.t \ = \ a.(s\restriction V).(t\restriction V)$$

◀

Restricting an action does not generally yield something that makes sense. For example, let x and y be distinct variables. Restricting $y := x + y + 1$ to $\{x\}$ yields $(\lambda s,t.\ (\aleph = s.x + \aleph + 1)\ \wedge\ (s\restriction\{x\} = t\restriction\{x\}))$.

What is perhaps more interesting is $\mathsf{skip}\restriction V$. It is an action that preserves the values of all variables in V. With it we can, for example, rewrite (6.3) to:

$$\mathsf{assign}.x.e \ = \ (\lambda s,t.\ t.x = e.s) \sqcap (\mathsf{skip}\restriction\{x\}^{\mathsf{c}}) \qquad (6.6)$$

where $\sqcap$ is the so-called synchronisation operator on actions, which is defined as follows:

Definition 6.4 SYNCHRONISATION OPERATOR *rINTER*

$$a \sqcap b \ = \ (\lambda s,t.\ a.s.t \wedge b.s.t)$$

◀

A simultaneous assignment can be defined as:

$$\begin{gathered} \mathsf{assign}_2.(x,y).(E_1,E_2) \\ = \\ (\lambda s,t.\ t.x = E_1.s)\ \sqcap\ (\lambda s,t.\ t.y = E_2.s)\ \sqcap\ (\mathsf{skip}\restriction\{x,y\}^{\mathsf{c}}) \end{gathered} \qquad (6.7)$$

An action is called *always enabled* if it is always ready to make a transition (even though it may be a skip transition).

Definition 6.5 ALWAYS ENABLED ACTION *ALWAYS_ENABLED*

$$\square_{\mathsf{En}} a \ = \ (\forall s :: (\exists t :: a.s.t))$$

◀

6.2 UNITY programs

Figure 3 displays an example of an UNITY program. The precise syntax will be given later.

The read and write sections declare, respectively, the read and write variables of the program. The init section describes the assumed initial state of the program. In the program

```
prog   Example
read   $\{a, x, y\}$
write  $\{x, y\}$
init   true
assign
       if $a = 0$ then $x := 1$ else skip
[]     if $a \neq 0$ then $x := 1$ else skip
[]     if $x \neq 0$ then $y, x := y + 1, 0$ else skip
```

Figure 3: The program Example

Example in Figure 3, the initial condition is true, which means that the program may start in any state. The assign section lists the actions (multiple-assignments statements) of the program, separated by the symbol $[\!]$.

Actions in a UNITY program are assumed to be *atomic*. An execution of a UNITY program starts in a state satisfying the initial condition and is an infinite and interleaved execution of its actions. In each step of the execution some action is selected non-deterministically and executed, so there is no ordering imposed on the execution of the actions. There is, however, one fairness condition:

In a UNITY execution, which is infinite, each action must be executed infinitely often, and hence cannot be ignored forever.

So in the program Example, eventually $x = 0$ will hold and if $M = y$, then eventually $M < y$ will hold. As far as UNITY concerns, the actions of this program can be implemented sequentially, fully parallel or anything in between, as long as the atomicity and the fairness condition of UNITY are met. Consequently, in constructing a UNITY program one is encouraged to concentrate on the 'real' problem, and not to worry about ordering and allocation of the actions, as such are considered to be implementation issues.

By now the reader should have guessed that a UNITY program P can be represented by a quadruple (A, J, V_r, V_w) where $A \subseteq \mathsf{Action}$ is a set consisting of P's actions, $J \in \mathsf{Pred}$ is a predicate describing the possible initial states of P, and $V_r, V_w \in \mathsf{Var}$ are sets containing P's read and write variables respectively. The set of all possible quadruples (A, J, V_r, V_w) shall be denoted by Uprog. Consequently, all UNITY programs are a member of this set, whereas, as will be made clear later, the converse is not necessarily true.

To access each component of an Uprog object, the destructors **a**, **ini**, **r**, and **w** are introduced. They satisfy the following property:

Theorem 6.6 Uprog DESTRUCTORS

$$P \in \mathsf{Uprog} \;=\; (P \;=\; (\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P))$$

In addition, the set of P's input variables , that is, the set containing the variables read by

P but not written by it, is denoted by $\mathbf{i}P$:

$$\mathbf{i}P = \mathbf{r}P \backslash \mathbf{w}P \tag{6.8}$$

6.2.1 The UNITY programming language

Below we present the syntax of UNITY programs that is used here. The syntax deviates slightly from the one in [CM88][6].

```
⟨Unity Program⟩ ::= prog   ⟨name of program⟩
                    read   ⟨set of variables⟩
                    write  ⟨set of variables⟩
                    init   ⟨predicate⟩
                    assign ⟨actions⟩
```

```
⟨actions⟩          ::= ⟨action⟩ | ⟨action⟩ ‖ ⟨actions⟩
⟨action⟩           ::= ⟨single action⟩ | (‖i : i ∈ V : ⟨actions⟩_i)
⟨single action⟩    ::= ⟨assignment⟩ | ⟨guarded action⟩
⟨assignment⟩       ::= ⟨variable-list⟩ := ⟨expr-list⟩
⟨variable-list⟩    ::= ⟨variable⟩ {, ⟨variable⟩}
⟨expr-list⟩        ::= ⟨expr⟩ {, ⟨expr⟩}
⟨expr⟩             ::= ⟨variable⟩ | ⟨expr⟩ ⟨ op_e⟩ ⟨expr⟩ | ⟨conditional-expr⟩
⟨conditional-expr⟩ ::= ⟨boolean-expr⟩ ⟹ ⟨expr⟩ ∼ ⟨expr⟩
⟨guarded action⟩   ::= if⟨boolean-expr⟩ then ⟨action⟩ else ⟨action⟩
                     | if⟨boolean-expr⟩ then ⟨action⟩
⟨boolean-expr⟩     ::= ⟨true⟩ | ⟨false⟩ | ⟨expr⟩ ⟨ op_c⟩ ⟨expr⟩
                     | ⟨boolean-expr⟩ ⟨ op_b⟩ ⟨boolean-expr⟩
```

Where op_e are the addition, subtraction, multiplication, etc ..., operators on numbers; op_c are the comparison operators (i.e. $<$, $>$, $=$, etc ...); and op_b are the boolean operators (i.e. $\wedge$, $\vee$, $\Rightarrow$, etc ...).

So *actions* is a list of *action*s separated by ‖, an *action* is either a single action or a set of indexed actions, and a single action is either an assignment such as $x := x + 1$ or a guarded action.

An assignment can assign to more than one variable at the same time. A variable may appear more than once in the left side of an assignment action. It is the programmer's responsibility to ensure for any such variable that all possible values that may be assigned to it in an action are identical.

A guarded action has the form:

```
if   g_1 then a_1
else g_2 then a_2
else g_3 then a_3
else ...
```

[6]We omit the always section and split the declare section into read and write parts

If more than one guard evaluate to true, then one is selected non-deterministically. If none of the guards evaluate to true, a guarded action behaves like skip. So, for example, the action "if $a \neq 0$ then $x := 1$ else skip" from the program Fizban can also be written as "if $a \neq 0$ then $x := 1$". Note that every guarded action which has the form:

$$\text{if } (a > 0) \text{ then } x := x + 1 \text{ else } x := x - 1$$

can be written as the single action:

$$x := (a > 0) \implies x + 1 \sim x - 1$$

There is no difference between the two, we just added the possibility of the latter notation for convenience.

6.2.2 The well-formedness of a UNITY program

Besides being an element of the language generated by the grammer which was presented in the previous section, the following requirements regarding the well-formedness of a UNITY program must be met:

semantical requirements

1. The actions of the program should be always-enabled

syntactical requirements

2. A write variable is also readable.
3. The actions of a program should only write to the declared write variables.
4. The actions of a program should only depend on the declared read variables.

These are perfectly natural requirements for a program. Most programs that are written shall satisfy them. In order to be able to capture these requirements into a predicate, we introduce two new notions, viz. *ignored-by* and *invisible-to*.

A set of variables is V *ignored-by* an action a, denoted by $V \not\leftarrow a$, if executing the action in any state does not change the values of these variables. Variables in V^c *may* however be written by a. The smallest set of variables which may be written by a is the set of variables which are actually written by a.

Definition 6.7 VARIABLES IGNORED-BY ACTION *IG_BY_DEF*

$$V \not\leftarrow a \;=\; (\forall s,t :: a.s.t \Rightarrow (s\restriction V = t\restriction V))$$

◀

The notion of *ignored-by* is used to formalise the second syntactical requirement from above. All actions of a program P only write to the declared write variables, if and only if these actions do not change the values of variables that are not declared as write variables, i.e. variables in the set $(\mathbf{w}P)^c$. So, the second syntactical requirement is precisely $(\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^c \not\leftarrow a)$.

A set of variables V is said to be *invisible-to* an action a, denoted by $V \not\twoheadrightarrow a$, if changing the values of the variables in V will not influence what a can do to the variables outside V, hence a only depends on the variables outside V.

Definition 6.8 VARIABLES INVISIBLE-TO ACTION *INVI_DEF*

$$V \not\twoheadrightarrow a$$
$$=$$
$$(\forall s, t, s', t' :: (s \restriction V^{\mathsf{c}} = s' \restriction V^{\mathsf{c}}) \wedge (t \restriction V^{\mathsf{c}} = t' \restriction V^{\mathsf{c}}) \wedge (s' \restriction V = t' \restriction V) \wedge a.s.t \;\Rightarrow\; a.s'.t')$$

◀

This notion of *invisible-to* is used to formalise the last syntactical requirement. All actions of a program P only depend on the declared read variables, if and only if changing the values of the variables that are not declared as read variables (i.e. variables in the set $(\mathbf{r}P)^{\mathsf{c}}$) will not influence a's result, hence a only depends on the variables outside $(\mathbf{r}P)^{\mathsf{c}}$ which are precisely the declared read variables. So the last syntactical requirement can be formalised by $(\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadrightarrow a)$.

Recall that any UNITY program is an object of type Uprog. Now a predicate Unity can be defined to express the well-formedness of an Uprog object. From here on, a "UNITY program" is an object satisfying the predicate Unity.

Definition 6.9 Unity

$$\begin{aligned} \mathsf{Unity}.P \;=\; & (\forall a : a \in \mathbf{a}P : \square_{\mathsf{En}} a) \;\wedge\; (\mathbf{w}P \subseteq \mathbf{r}P) \;\wedge \\ & (\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a) \;\wedge\; (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadrightarrow a) \end{aligned}$$

◀

6.3 UNITY logic

In UNITY, specifications are expressed using normal predicate logic plus three additional logical binary operators (unless, ensures, and $\mapsto$ ("leads-to")), that facilitate reasoning about program behaviour.

The discussion in Section 6.2 revealed that an execution of a UNITY program never, in principle, terminates. Therefore we focus on the behaviour of a program *during* its execution. Two aspects will be considered: progress and safety. A *progress property* of a program expresses what the program is expected to eventually realize. For example, if a message is sent through a routing system, a progress property may state that eventually the message will be delivered to its destination. A *safety property*, on the other hand, tells us what the program should not do: for example, that the message is only to be delivered to its destination, and not to any other computer.

In UNITY safety behaviour is described by an operator called unless (definition 6.10).[7] Intuitively, ${}_P\vdash p$ unless q implies that once p holds during an execution of P, it remains to hold at least until q holds. Figure 4 may be helpful. Note that this interpretation gives no information whatsoever about what p unless q means if p never holds during an execution.

[7] In the sequel, P, Q, and R will range over UNITY programs; a, b, and c over Action; and p, q, r, s, J and K over Pred.

$$\begin{tikzcd} p \text{ unless } q : & p \wedge \neg q \arrow[loop above] \arrow[r] & q \end{tikzcd} \qquad \begin{tikzcd} p \text{ ensures } q : & p \wedge \neg q \arrow[loop above] \arrow[r, "!"] & q \end{tikzcd}$$

Figure 4: unless and ensures. The predicates $p \wedge \neg q$ and q define sets of states. The arrows depict possible transitions between the two sets of states. The arrow marked with ! is a guaranteed transition.

◀

By the fairness condition of UNITY, an action cannot be continually ignored. Once executed, it may induce some progress. For example, the execution of the action in the program Example (figure 3) shall establish $x = 0$ regardless when it is executed. This kind of single-action progress is described by an operator called ensures (definition 6.11). ${}_P\vdash p$ ensures q encompasses p unless q, and adds that there should also exist an action that can, and because of the fairness assumption of UNITY, will establish q.

Definition 6.10 UNLESS

$$ {}_P\vdash p \text{ unless } q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\}) $$

Definition 6.11 ENSURES

$$ {}_P\vdash p \text{ ensures } q \;=\; ({}_P\vdash p \text{ unless } q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{q\}) $$

◀

To illustrate these definitions, consider again the program Example in Figure 3. It has the following assign section:

```
   if $a = 0$ then $x := 1$
[] if $a \neq 0$ then $x := 1$
[] if $x \neq 0$ then $y, x := y + 1, 0$
```

The following properties are satisfied:

$$ {}_{\text{Example}}\vdash \quad (a = X) \text{ unless false} \tag{6.9} $$
$$ {}_{\text{Example}}\vdash \quad \text{true unless } (x = 1) \tag{6.10} $$
$$ {}_{\text{Example}}\vdash \quad (a = 0) \text{ ensures } (x = 1) \tag{6.11} $$
$$ {}_{\text{Example}}\vdash \quad (a \neq 0) \text{ ensures } (x = 1) \tag{6.12} $$

If (6.9) holds for any X then it states that Example cannot change the value of a. In (6.10) we find an example of a property that trivially holds in any program (the reader can check it by unfolding the definition of unless). (6.11) and (6.12) describe single-action progress from, respectively $a = 0$ and $a \neq 0$ to $x = 1$.

Theorem 6.12 ensures INTRODUCTION *ENSURES_IMP_LIFT*

$$P: \frac{[p \Rightarrow q]}{p \ \mathsf{ensures}\ q}$$

Theorem 6.13 ensures POST-WEAKENING *ENSURES_CONSQ_WEAK*

$$P: \frac{(p \ \mathsf{ensures}\ q) \ \wedge \ [q \Rightarrow r]}{p \ \mathsf{ensures}\ r}$$

Theorem 6.14 ensures PROGRESS SAFETY PROGRESS (PSP) *ENSURES_PSP*

$$P: \frac{(p \ \mathsf{ensures}\ q) \ \wedge \ (r \ \mathsf{unless}\ s)}{p \wedge r \ \mathsf{ensures}\ (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

Theorem 6.15 ensures CONJUNCTION *ENSURES_CONJ*

$$P: \frac{(p \ \mathsf{ensures}\ q) \ \wedge \ (r \ \mathsf{ensures}\ s)}{p \wedge r \ \mathsf{ensures}\ (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

Figure 5: Some properties of ensures

◀

Properties of the form ${}_P\vdash p \ \mathsf{unless\ false}$ are called *stable* properties, which are very useful properties because they express that once p holds during any execution of P, it will remain to hold forever. Because of their importance a separate abbreviation is defined in definition 6.16 below.

Definition 6.16 STABLE PREDICATE

$${}_P\vdash \circlearrowleft p \ = \ {}_P\vdash p \ \mathsf{unless\ false}$$

◀

${}_P\vdash \circlearrowleft p$ is pronounced "*p is stable in P*" and p is called a *stable predicate.* Notice that $\circlearrowleft$ can also be defined as follows:

$${}_P\vdash \circlearrowleft p \ = \ (\forall a : a \in \mathbf{a}P : \{p\}\ a\ \{p\}) \tag{6.13}$$

Consequently, if p holds initially and is stable in program P, it will hold throughout any execution of P, and hence it is an *invariant* of P, which shall be denoted by ${}_P\vdash \Box p$. As a result, we have the following definition:

Definition 6.17 INVARIANT

$${}_P\vdash \Box J \ = (\mathbf{ini}P \Rightarrow J \wedge {}_P\vdash \circlearrowleft J)$$

◀

Figures 6 and 5 list some interesting properties of unless , $\circlearrowleft$ and ensures . These properties are taken from [CM88], and there are more to be found there. Most of the properties

Theorem 6.18 unless Introduction *UNLESS_IMP_LIFT1, UNLESS_IMP_LIFT2*

$$P: \frac{[p \Rightarrow q] \;\vee\; [\neg p \Rightarrow q]}{{}_P\!\vdash p \textsf{ unless } q}$$

Theorem 6.19 unless Post-weakening *UNLESS_CONSQ_WEAK*

$$P: \frac{(\,{}_P\!\vdash p \textsf{ unless } q) \;\wedge\; [q \Rightarrow r]}{{}_P\!\vdash p \textsf{ unless } r}$$

Theorem 6.20 unless Simple Conjunction *UNLESS_SIMPLE_CONJ*

$$P: \frac{(\,{}_P\!\vdash p \textsf{ unless } q) \;\wedge\; (\,{}_P\!\vdash r \textsf{ unless } s)}{{}_P\!\vdash p \wedge r \textsf{ unless } q \vee s}$$

Theorem 6.21 unless Simple Disjunction *UNLESS_SIMPLE_DISJ*

$$P: \frac{(\,{}_P\!\vdash p \textsf{ unless } q) \;\wedge\; (\,{}_P\!\vdash r \textsf{ unless } s)}{{}_P\!\vdash p \vee r \textsf{ unless } q \vee s}$$

Theorem 6.22 $\circlearrowright$ Conjunction *STABLE_CONJ*

$$P: \frac{(\,{}_P\!\vdash \circlearrowright p) \;\wedge\; (\,{}_P\!\vdash \circlearrowright q)}{{}_P\!\vdash \circlearrowright (p \wedge q)}$$

Theorem 6.23 $\circlearrowright$ Disjunction *STABLE_DISJ*

$$P: \frac{(\,{}_P\!\vdash \circlearrowright p) \;\wedge\; (\,{}_P\!\vdash \circlearrowright q)}{{}_P\!\vdash \circlearrowright (p \vee q)}$$

Figure 6: Some properties of unless and $\circlearrowright$.

can be derived from the definitions 6.11 and 6.10. Some of the properties (Theorems 6.19 and 6.21) look similar to some well known rules for Hoare triples[8].

As a notational convention: *if it is clear from the context which program P is meant, we often omit it from a formula.* For example we may write p unless q to mean ${}_P\!\vdash p$ unless q. Also, for laws we write, for example:

$$P: \frac{\ldots(p \textsf{ unless } q)\ldots}{r \textsf{ unless } s} \quad \text{to abbreviate:} \quad \frac{\ldots(\,{}_P\!\vdash p \textsf{ unless } q)\ldots}{{}_P\!\vdash r \textsf{ unless } s}$$

The ensures operator is still too restricted to describe progress (it only describes single action progress). Intuitively, progress seems to have transitivity and disjunctivity properties. For example, if a system can progress from p to q and from q to r, then it can progress from p to r (transitivity). If it can progress from p_1 to q and p_2 to q, then from either p_1 or p_2 it

[8]Note though that the pre-condition strengthening principle of Hoare triples does not apply to unless and ensures

can progress to q (disjunctivity). As a more general progress operator we can therefore take the smallest closure of ensures which is transitive and disjunctive. The resulting operator is the *leads-to* operator, denoted by $\mapsto$:

Definition 6.24 Leads-to
$(\lambda p, q.\ {}_P\vdash p \mapsto q)$ is defined as the smallest relation R satisfying:

i. $$\frac{{}_P\vdash p \ \mathsf{ensures}\ q}{J\ {}_P\vdash R.p.q}$$

ii. $$\frac{J\ {}_P\vdash R.p.q \wedge J\ {}_P\vdash R.q.r}{J}\ {}_P\vdash R.p.r$$

iii. $$\frac{(\forall i : i \in W : J\ {}_P\vdash R.(p_i).q)}{J\ {}_P\vdash R.(\exists i : i \in W : p_i).q}$$

◀

Obvious properties of $\mapsto$ is that it satisfies ***i***, ***ii***, and ***iii*** above. ${}_P\vdash p \mapsto q$ implies that that if p holds during an execution of P, then eventually q will hold, so it corresponds with our intuitive notion of progress.

6.4 Extended UNITY

As mentioned in section 3 Prasetya extended the UNITY programming logic with two logical operators, that concerned compositionality and self-stabilisation. These two operators will be described in the two sections below.

6.4.1 Compositionality

A consequence of the absence of ordering in the execution of a UNITY program is that the parallel composition of two programs can be modelled by simply merging the variables and actions of both programs. In UNITY parallel composition is denoted by $[\!]$. In [CM88] the operator is also called *program union.*

Definition 6.25 Parallel Composition

$$P[\!]Q \ = \ (\mathbf{a}P \cup \mathbf{a}Q, \mathbf{ini}P \wedge \mathbf{ini}Q, \mathbf{r}P \cup \mathbf{r}Q, \mathbf{w}P \cup \mathbf{w}Q)$$

◀

Parallel composition is reflexive, commutative, and associative. It has a unit element, namely $(\emptyset, \mathsf{true}, \emptyset, \emptyset)$ (although this is not a well-formed UNITY program).

To illustrate, the program Example in Figure 3 can be composed with the program below:

```
prog    TikTak
read    {a}
write   {x}
init    true
assign  if a = 0 then a := 1 [] if a ≠ 0 then a := 0
```

Theorem 6.26 unless COMPOSITIONALITY *UNLESS_PAR_i*

$$({}_{P}\vdash p \;\mathsf{unless}\; q) \;\wedge\; ({}_{Q}\vdash p \;\mathsf{unless}\; q) \;\;=\;\; ({}_{P[]Q}\vdash p \;\mathsf{unless}\; q)$$

Theorem 6.27 $\circlearrowleft$ COMPOSITIONALITY *STABLE_PAR_i*

$$({}_{P}\vdash \circlearrowleft J) \;\wedge\; ({}_{Q}\vdash \circlearrowleft J) \;\;=\;\; ({}_{P[]Q}\vdash \circlearrowleft J)$$

Theorem 6.28 ensures COMPOSITIONALITY *ENSURES_PAR*

$$\frac{({}_{P}\vdash p \;\mathsf{ensures}\; q) \;\wedge\; ({}_{Q}\vdash p \;\mathsf{unless}\; q)}{{}_{P[]Q}\vdash p \;\mathsf{ensures}\; q}$$

Figure 7: Some theorems describing the properties of parallel compositions.

The resulting program consists of the following actions (the else skip part of the actions in Example will be dropped, which is, as remarked in Section 6.2.1, allowed):

$$\begin{array}{ll}
a_0: & \mathsf{if}\; a = 0 \;\mathsf{then}\; a := 1 \\
a_1: & \mathsf{if}\; a \neq 0 \;\mathsf{then}\; a := 0 \\
a_2: & \mathsf{if}\; a = 0 \;\mathsf{then}\; x := 1 \\
a_3: & \mathsf{if}\; a \neq 0 \;\mathsf{then}\; x := 1 \\
a_4: & \mathsf{if}\; x \neq 0 \;\mathsf{then}\; y, x := y + 1, 0
\end{array}$$

Whereas in Example $x \neq 0$ will always hold somewhere in the future, the same cannot be concluded for Example [] TikTak. Consider the execution sequence $(a_0; a_2; a_1; a_3; a_4)*$, which is a fair execution and therefore a UNITY execution. In this execution, the assignment $x := 1$ will never be executed. If initially $x \neq 1$ this will remain so for the rest of this execution sequence.

In designing a program, one often splits the program into smaller components (modularity principle). It would therefore be desirable if one could also decompose a global specification into smaller specifications of component programs. This would enable us to design each component program in isolation. In order to enable these kind of decompositions laws of the form

$$\frac{(P \;\mathsf{sat\; spec1}) \;\wedge\; (Q \;\mathsf{sat\; spec2})}{P \otimes Q \;\mathsf{sat}\; (\mathsf{spec1} \oplus \mathsf{spec2})} \tag{6.14}$$

are required. P and Q are programs, $\otimes$ is some kind of program composition, and spec1 and spec2 are specifications. This law states a *compositionality property* of $\oplus$ with respect to the program composition $\otimes$. Such properties enable us to split the specification of a composite program $P \otimes Q$ into the specifications of P and Q. In particular, we are interested in the case where $\otimes$ is some form of parallel composition. An advantage of having compositional properties is that they often significantly reduces the amount of proof obligations. In figure 7 are several compositionality properties of unless , $\circlearrowleft$ and ensures .

In particular, notice how Theorem 6.28 describes the condition in which progress by ensures can be preserved by parallel composition. The theorem does not apply to progress by $\mapsto$ though. The extension presented in [Pra95] is a variant of $\mapsto$. It is called the *reach* operator, and is denoted by $\rightarrowtail$. In the rest of this section the reach operator is defined and the most important compositionality laws on this operator are presented.

Definition 6.29 Reach Operator
$(\lambda p, q. J\ {}_P\!\vdash p \rightarrowtail q)$ is defined as the smallest relation R satisfying:

i. $$\frac{p, q \in \mathsf{Pred}.(\mathbf{w}P)\ \wedge\ ({}_P\!\vdash \circlearrowright J)\ \wedge\ ({}_P\!\vdash J \wedge p \ \mathsf{ensures}\ q)}{J\ {}_P\!\vdash R.p.q}$$

ii. $$\frac{J\ {}_P\!\vdash R.p.q \wedge J\ {}_P\!\vdash R.q.r}{J\ {}_P\!\vdash R.p.r}$$

iii. $$\frac{(\forall i : i \in W : J\ {}_P\!\vdash R.(p_i).q)}{J\ {}_P\!\vdash R.(\exists i : i \in W : p_i).q}$$

where W is assumed to be non-empty.

--- ◀

Intuitively, $J\ {}_P\!\vdash p \rightarrowtail q$ means that J is stable in P and that P can progress from $J \wedge p$ to q. J in a certain sense captures the assumptions made about the environment under which progress is ensured. In addition, the type of p and q is restricted: they are predicates over $\mathbf{w}P \rightarrow \mathsf{Val}$ (the part of state space restricted to the write variables of P). Since P can only write to its write variables, it will make progress only on these variables. Therefore it is reasonable to restrict the type of p and q as above. Whatever values variables outside $\mathbf{w}P$ may have will remain stable then, and can consequently be specified in J. This division turns out to yield a compositional progress operator.

Figure 8 list some other interesting properties of $\rightarrowtail$. Again, as a notational convention: *if it is clear from the context which program P or which stable predicate J are meant, we often omit them from an expression.* For example we may write ${}_P\!\vdash p \rightarrowtail q$ or even simply $p \rightarrowtail q$ to mean $J\ {}_P\!\vdash p \rightarrowtail q$. Also, for laws we write, for example:

$$P, J : \frac{\ldots (p\ \mathsf{unless}\ q) \ldots}{r \rightarrowtail s} \quad \text{to abbreviate:} \quad \frac{\ldots ({}_P\!\vdash p\ \mathsf{unless}\ q) \ldots}{J\ {}_P\!\vdash r \rightarrowtail s}$$

The operator $\rightarrowtail$ Introduction states that if $p \Rightarrow q$ holds, then it is trivial that any program P can progress from p to q. Theorem $\rightarrowtail$ Disjunction asserts that $\rightarrowtail$ is disjunctive at its left and right operands. The $\rightarrowtail$ Substitution law expounds that, like Hoare triples, we can strengthen pre-conditions and weaken post-conditions. The PSP law indicates that a safety property (unless) of a program can influence its progress. And finally $\rightarrowtail$ Bounded Progress gives the well-founded induction principle for the reach operator, that is, if, from p, a program can progress to q, or else it maintains p while decreasing the value of m with respect to a well-founded[9] ordering $\prec$, then, since $\prec$ is well-founded, it is not possible to keep decreasing m, and hence eventually q will be established.

To give the most important compositionality laws on $\rightarrowtail$, it is necessary to define a special case of unless .

[9]An ordering $\prec \in A \rightarrow A \rightarrow \mathbb{B}$ is said to be *well-founded* if it is not possible to construct an infinite sequence of ever decreasing values in A. That is, $\ldots, x_2 \prec x_1 \prec x_0$ is not possible.

Theorem 6.30 $\rightarrowtail$ INTRODUCTION *REACH_ENS_LIFT,REACH_IMP_LIFT*

$$P, J: \ \frac{p, q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_P\vdash \circlearrowleft J) \ \wedge \ [J \wedge p \Rightarrow q]}{J \ {}_P\vdash p \rightarrowtail q}$$

Theorem 6.31 $\rightarrowtail$ DISJUNCTION *REACH_DISJ*

$$P, J: \ \frac{(J \ {}_P\vdash p \rightarrowtail q) \ \wedge \ (J \ {}_P\vdash r \rightarrowtail s)}{J \ {}_P\vdash p \vee r \rightarrowtail q \vee s}$$

Theorem 6.32 $\rightarrowtail$ SUBSTITUTION *REACH_SUBST*

$$P, J: \ \frac{\begin{array}{c} p, s \in \mathsf{Pred}.(\mathbf{w}P) \\ [J \wedge p \Rightarrow q] \ \wedge \ (J \ {}_P\vdash q \rightarrowtail r) \ \wedge \ [J \wedge r \Rightarrow s] \end{array}}{J \ {}_P\vdash p \rightarrowtail s}$$

Theorem 6.33 $\rightarrowtail$ PROGRESS SAFETY PROGRESS (PSP) *REACH_PSP*

$$P, J: \ \frac{r, s \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_P\vdash r \wedge J \ \mathsf{unless} \ s) \wedge \ (J \ {}_P\vdash p \rightarrowtail q)}{J \ {}_P\vdash p \wedge r \rightarrowtail (q \wedge r) \vee s}$$

Theorem 6.34 $\rightarrowtail$ BOUNDED PROGRESS *REACH_WF_INDUCT*

$$P, J: \ \frac{q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (\forall M :: p \wedge (m = M) \rightarrowtail (p \wedge (m \prec M)) \vee q)}{p \rightarrowtail q}$$

With $\prec$ a well-founded ordering over A and $m \in \mathsf{State} \rightarrow A$ is a bound function.

Figure 8: Some properties of $\rightarrowtail$.

◀

Definition 6.35 unless_V
Let V be a set of variables:

$$_Q\vdash p \ \mathsf{unless}_V \ q \ = \ (\forall X :: \ _Q\vdash p \wedge (\lambda s. \ (\forall v : v \in V : s.v = X.v)) \ \mathsf{unless} \ q)$$

◀

In particular, if $V = \mathbf{r}P \cap \mathbf{w}Q$ (hence V are the 'border' variables from Q to P), $_Q\vdash p \ \mathsf{unless}_V \ q$ means that under condition p, Q *cannot* influence P without establishing q. So, for example, $_Q\vdash p \ \mathsf{unless}_V \ \mathsf{false}$ means that Q cannot influence P as long as p holds; $_Q\vdash \mathsf{true} \ \mathsf{unless}_V \ q$ means that Q always marks its interference to P by establishing q.

Let $V = \mathbf{r}P \cap \mathbf{w}Q$. Suppose under condition r, Q cannot influence P without establishing s (i.e. $_Q\vdash r \ \mathsf{unless}_V \ s$). Hence Q cannot destroy any progress in P without establishing q. This principle —or actually, a more general version thereof— is formulated by the law below. It is called the *Singh* law.

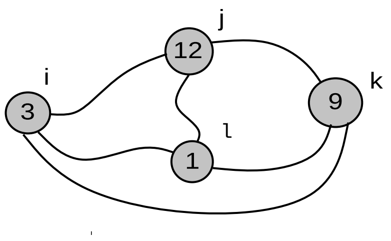

Figure 9: A simple network.

Theorem 6.36 SINGH LAW *REACH_SINGH*

$$\frac{r, s \in \mathsf{Pred}.\mathbf{w}(P \| Q) \;\wedge\; ({}_Q\vdash \circlearrowleft J) \;\wedge\; ({}_Q\vdash J \wedge r \;\mathsf{unless}_V\; s) \;\wedge\; (J \;{}_P\vdash p \rightarrowtail q)}{J \;{}_{P\|Q}\vdash p \wedge r \rightarrowtail q \vee \neg r \vee s}$$

where $V = \mathbf{r}P \cap \mathbf{w}Q$.

A corollary of Singh Law is given below. Compare it with the compositionality law of $\mathsf{ensures}$ (Theorem 6.28).

$$\frac{({}_Q\vdash \circlearrowleft J) \;\wedge\; ({}_Q\vdash J \wedge p \;\mathsf{unless}_V\; q) \;\wedge\; ({}_P\vdash J \wedge p \;\mathsf{unless}\; q) \;\wedge\; (J \;{}_P\vdash p \rightarrowtail q)}{J \;{}_{P\|Q}\vdash p \rightarrowtail q}$$

$\rightarrowtail$ also satisfies a very nice principle called the *Transparency* principle:

Theorem 6.37 TRANSPARENCY LAW

$$\frac{(\mathbf{w}P \cap \mathbf{w}Q = \emptyset) \;\wedge\; ({}_Q\vdash \circlearrowleft J) \;\wedge\; (J \;{}_P\vdash p \rightarrowtail q)}{J \;{}_{P\|Q}\vdash p \rightarrowtail q}$$

So, in a network of components with disjoint write variables, any progress $J \;{}_P\vdash p \rightarrowtail q$ in a component P will be preserved if all components respect the stability of J. A network of programs with disjoint write variables occurs quite often in practice. For example a network with write-disjoint components can model a network of programs that communicate using channels.

6.4.2 Self-stabilisation

Imagine a network of processes as in Figure 9, which is fully connected. Each node has a label (i.e a name) which in Figure 9 is printed above the node. Each node also contains data, which in Figure 9 consist of one number. Suppose that the labels are also numbers that come from a collection called Labels. Furthermore, let V be a function that given a

```
prog   Sort
read   $\{Vi|\, i \in \mathsf{Labels}\}$
write  $\{Vi|\, i \in \mathsf{Labels}\}$
init   true
assign($[\!]i,j : (i,j \in \mathsf{Labels}) \wedge (i \leq j) :: (V\,i),(V\,j) := \mathsf{min}(V\,i, V\,j),\ \mathsf{max}(V\,i, V\,j)$)
```

Figure 10: A sorting program

label l, returns the data value that resides in the node labelled by l. Then this network being sorted can be defined as:

$$\mathsf{Sorted} = \forall i,j \in \mathsf{Labels}\text{: } i \leq j \Rightarrow (V\,i) \leq (V\,j)$$

An algorithm that establishes this property for any network as described above is displayed in Figure 10. This algorithm exploits an "swap-out-of-order-values" strategy, that is if the values of two nodes are out-of-order according to the definition above then the values of these nodes are swapped. The functions min and max determine the minimum and maximum of their two inputs respectively.

Notice that the program has an initial condition true, which means that it will work correctly no matter in which state it is started. Consequently, if during its execution an external agent interferes with it and tampers with the data values of the nodes, this can be considered as if the program is re-started in a new initial state. Since the program works correctly regardless of its initial state, it will also do so in this new situation. In addition, once the network is sorted, the situation will be maintained forever (or until the next interference by the environment). Such properties are called *self-stabilising* properties and are clearly very useful due to their failure-recovering behaviour. Self-stabilisation makes it possible to write programs in an unstable environment, i.e. an environment that may produce transient errors, or undergo a spontaneous reconfiguration which affect the consistency of the variables upon which the program depends. Self-stabilisation is, however, a strong design goal. Perhaps too strong as it may be either too difficult to achieve or only be achievable at the expense of other goals. As failures are not always arbitrary it is useful to consider recovery from a restricted set of failures. The notion of self-stabilisation can therefore be generalised to express this kind of weaker recovery. There is, however, another reason to make such a generalisation: it may yield more attractive and useful calculational laws. The definition for this generalised notion of self-stabilisation, which is also called convergence, is given in definition 6.38.

Definition 6.38 CONVERGENCE *CON*

$$J\ {}_P\!\vdash p \rightsquigarrow q$$
$$=$$
$$q \in \mathsf{Pred.}(\mathbf{w}P)\ \wedge\ (\exists q' :: (J\ {}_P\!\vdash p \rightarrowtail q' \wedge q)\ \wedge\ ({}_P\!\vdash \circlearrowleft (J \wedge q' \wedge q)))$$

So, $J\ {}_{P}\!\vdash p \rightsquigarrow q$ implies that under the stability of J, from p the program P will eventually find itself in a situation where q holds and will remain to hold. $J \vdash p \rightsquigarrow q$ is pronounced "given the stability of J, from p, P converges to q". The reader might wonder why the q' is necessary. Well, suppose that a program P can progress from p to q. However, P may not remain in q immediately after the first time q holds. Instead, P may need several iterations before it finally remains within q. This can be encoded by requiring P to converge to a stronger (than q) predicate. This predicate does not need to be fully described. It suffices to know that it implies q[10].

Figure 11 lists properties of convergence. Notice that what distinguishes $\rightsquigarrow$ from $\rightarrowtail$ is that the first is conjunctive (Theorem 6.45) and the second is not. The most important properties, however, are shared by both. For example Bounded Progress, a consequence of the well-founded induction principle, is applicable to either $\rightsquigarrow$ and $\rightarrowtail$. This is very pleasant, since well-founded induction is a standard technique to prove termination, moreover the $\mapsto$ version of the Bounded Progress law appears in [CM88] as a standard technique to prove progress. An even stronger induction principle, called Round Decomposition (Theorem 6.47), exists for convergence. The principle exploits the conjunctivity of convergence — a property which is not enjoyed by $\rightarrowtail$ or $\mapsto$. Imagine a tree of processes. Each process collects the results of its sons, makes its own progress and then passes the result to its father. Consider a node n with sons l and m. Suppose l and m make progress to, respectively, q_l and q_m. The progress of l and m does not however combine to $q_l \wedge q_m$ because progress is not conjunctive. However, if processes l and m converge to q_l and q_m then $q_l \wedge q_m$ can be concluded. The process n may therefore make a stronger assumption to establish its own progress. If each process n has the following convergence property:

$$(\forall m : \text{"}m \text{ is a proper descendant of } n\text{"} : q_m) \rightsquigarrow q_n$$

then by applying tree induction one may conclude that eventually the system will converge to $(\forall m : m \in A : q_m)$, where A is the set of all processes in the tree. The principle applies not only to trees, but to any structure that can be ordered by a well-founded relation.

Theorem 6.47 Round Decomposition *CON_BY_sWF_i*

For all finite and non-empty sets A and all well-founded relations $\prec \in A \times A$:

$$P : \frac{(\circlearrowright J) \ \wedge \ (\forall n : n \in A : J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n)}{J \vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : q.n)}$$

◀

The name Round Decomposition can be explained as follows.

In sequential programming, we have a law for decomposing a while-loop specification into the specifications of the loop's guard and body. In fact, the whole sequential programming relies basically on loops. In UNITY we do not have loops. At least not explicitly. Recall that any execution of a UNITY program is infinite and that each action must be executed infinitely often. In this sense a UNITY program is actually one large while-loop. Such a loop can be viewed in terms of rounds. Each iteration step makes the system advance to the next round, until the final round is reached. In a sequential system the rounds are totally ordered. This, however, does not have to be the case in a distributed system,

[10]This notion of convergence is what by Burns, Gouda, and Miller called *pseudo-stabilisation* [BGM90].

Theorem 6.39 CONVERGENCE IMPLIES PROGRESS *CON_IMP_REACH*

$$P, J : \frac{p \rightsquigarrow q}{p \rightarrowtail q}$$

Theorem 6.40 $\rightsquigarrow$ INTRODUCTION *CON_ENSURES_LIFT, CON_IMP_LIFT*

$$P, J : \frac{\begin{array}{c} p, q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \ \wedge \ (\circlearrowleft (J \wedge q)) \\ [p \wedge J \Rightarrow q] \ \vee \ (p \wedge J \ \mathsf{ensures}\ q) \end{array}}{p \rightsquigarrow q}$$

Theorem 6.41 $\rightsquigarrow$ SUBSTITUTION *CON_SUBST*

$$P, J : \frac{[J \wedge p \Rightarrow q] \ \wedge \ [J \wedge r \Rightarrow s] \ \wedge \ p, s \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (q \rightsquigarrow r)}{p \rightsquigarrow s}$$

Theorem 6.42 $\rightsquigarrow$ REFLEXIVITY *CON_REFL*

$$P, J : \frac{p \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \ \wedge \ (\circlearrowleft (J \wedge p))}{p \rightsquigarrow p}$$

Theorem 6.43 $\rightsquigarrow$ TRANSITIVITY *CON_TRANS*

$$P, J : \frac{(p \rightsquigarrow q) \ \wedge \ (q \rightsquigarrow r)}{p \rightsquigarrow r}$$

Theorem 6.44 $\rightsquigarrow$ DISJUNCTION *CON_DISJ*

$$P, J : \frac{(\forall i : i \in W : p.i \rightsquigarrow q.i)}{(\exists i : i \in W : p.i) \rightsquigarrow (\exists i : i \in W : q.i)} \quad \text{if } W \neq \emptyset$$

Theorem 6.45 $\rightsquigarrow$ CONJUNCTION *CON_CONJ*
For all *non-empty* and *finite* sets W:

$$P, J : \frac{(\forall i : i \in W : p.i \rightsquigarrow q.i)}{(\forall i : i \in W : p.i) \rightsquigarrow (\forall i : i \in W : q.i)}$$

Theorem 6.46 $\rightsquigarrow$ BOUNDED PROGRESS *CON_WF_INDUCT*

$$P, J : \frac{(q \rightsquigarrow q) \ \wedge \ (\forall M :: p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q)}{p \rightsquigarrow q}$$

With $\prec$ a well-founded relation over A and $m \in \mathsf{State} \rightarrow A$ is a bound function.

Figure 11: Some basic properties of $\rightsquigarrow$.

◀

although well-foundedness will still be required. In sequential programming, invariants are used to specify the obligation of each iteration step. Imagine a distributed system that iterates along a (finite and well-founded) ordering $\prec$ over the domain of rounds to establish $(\forall n : n \in A : q_n)$. Theorem 6.47 states that it suffices to have:

$$J \wedge (\forall m : m \prec n : q_m) \vdash \mathsf{true} \rightsquigarrow q.n$$

for each round n. The above specifies the obligation of each round. The predicate $J \wedge (\forall m : m \prec n : q_m)$ can be considered some sort of loop invariant.

7 Embedding UNITY in HOL

By embedding a logic into a theorem prover the theorem prover is extended by all definitions required by the logic, and all basic theorems of the logic are made available —either by proving them or declaring them as axioms[11]. There are two kinds of embedding: the so-called *deep embedding* and *shallow embedding.* In a deep embedding, a logic is embedded down to the syntax level, whereas in a shallow embedding only the semantic, or model, of the logic needs to be embedded. A deep embedding is more trustworthy, but basically more difficult as we have to take the grammar of well-formed formulae in the logic into account. In [Pra95] a shallow embedding of the whole UNITY logic and all extensions (which were discussed in the previous section) is given in HOL. Basically, because whole UNITY is available in HOL, a program derivation can now take place entirely within HOL. Still, because of the flexibility of pencil and paper, sometimes it is very helpful that *one does the derivation by hand first, either in detail or only sketchy, and later verify it with HOL.* This current section discusses how UNITY programs are represented in HOL.

In section 6.2 a UNITY program was characterised as a quadruple (A, J, V_r, V_w) where A is a set of actions, J is a state-predicate describing allowed initial states, V_r is a set of variables intended to be the read variables of the program, and V_w those to be written. The universe of all variables are represented in HOL by a polymorphic type `*var` , and the universe of all values that these variables can take are represented by `*val` . For a concrete program one may want to, for instance, use strings to represent variables, and natural numbers as the domain of values. In this case the polymorphic types `*var` and `*val` simply have to be instantiated to `string` and `num` respectively. In practice, people often want to have programs in which the variables have different types —and, which may include sophisticated types such as functions or trees. In other words, a multi-typed universe of values is desired. This is possible, albeit not pleasant, as our universe of values is the type `*val` and hence multi-typed values have to be encoded *within* `*val`. For example, if one wishes both boolean and integer valued program variables, a new type must be defined:

```
define_type = `int_bool_DEF` `int_bool = INT int | BOOL bool` ;;
```

The above defines a new type called `int_bool`. A member of this type has the form `INT n` or `BOOL b` where `n` has the type `int` and `b` the type `bool`. Hence, instantiating `*val` with this type makes it possible to accommodate both bool and integers values[12].

The universe of program-states is represented by `State`:

[11]However, adding axioms, as remarked before, is not a recommended practice.

[12]This means however that all normal operations on integers and bool now have to be lifted to work on

```
let State = ":*var -> *val" ;;
```

7.1 Predicates and Predicate Operators

State-predicates are mapping from program-states to $\mathbb{B}$. The universe of state-predicates is represented by `Pred`:

```
let Pred = ":^State -> bool" ;;
```

An example is `(\s:^Pred. (s x = f (s y)))` which is a predicate that characterises those program-states s satisfying $s.x = f.(s.y)$.

This predicate is usually and conveniently denotes as, $x = f.y$. This notation is overloaded in several places. Since this kind of overloading is not possible in HOL, basically everything has to be made explicit using λ abstractions as above. Frequently used operators at the predicate-level, such as $\neg$, $\wedge$, $\vee$, and so on, are defined at the state-predicate-level by using the functions given in definition below.

HOL-definition 7.1

```
|- TT            = (\s. T)                 |- FF            = (\s. F)
|- (NOT p)       = (\s. ~p s)              |- (p AND q)     = (\s. p s /\ q s)
|- (p OR q)      = (\s. p s \/ q s)        |- (p IMP q)     = (\s. p s ==> q s)
|- (p EQUAL q)   = (\s. (p s = q s))       |- (!!i::P. Q i) = (\s. (!i::P. Q i s))
|- (??i::P. Q i) = (\s. (?i::P. Q i s))    |-  |== p        = (!s. p s)
```

◀

So, for instance the predicate that is usually denoted by $(x = f.y) \wedge q$ will be denoted in HOL as `(\s. s x = f (s y)) AND q`. Notice that `(!!i::P. Q i)` and `(??i::P. Q i)` above denote $(\forall i : P.i : Q.i)$ and $(\exists i : P.i : Q.i)$ at the predicate level.

Predicate confinement (definition 6.2) is defined as follows in HOL:

HOL-definition 7.2

```
|- !V A x. (V Pj A)x = (A x => V x | Nov)
|- !A p.  A CONF p = (!s t. (s Pj A = t Pj A) ==> (p s = p t))
```

◀

Where `Nov` is a HOL constant representing $\aleph$. Note that `A x` means $x \in A$, in other words, sets are represented by predicates that given some value return true if this value is an element of the set, and false otherwise. The reason for doing this is that HOL is nimbler with predicates than with sets.

7.2 Actions

An action is defined as a relation[13] on program-states, describing possible transitions the action can make. The universe of actions can be represented by `Action` in HOL:

this new type, which is quite tedious. There is another way to represent a program, in which differently typed variables are easy to represent. But this representation has its own problems too. See for instance [BvW90, Lån94]

[13]Some people prefer to use functions instead of relations. If functions are used, then actions are deterministic.

```
let Action = ":^State -> ^State -> bool" ;;
```

For instance, the simultaneous assignment action $x, y := E_1, E_2$ from (6.7) can be defined as follows in HOL:

```
let Assign2_DEF = new_definition
  (`Assign2_DEF`,
  "(Assign2 (x,y) (E1, E2)):^Action
     = (\s t. (t x = E1 s) /\ (t y = E2 s))
          rINTER
       (SKIP a_Pj (\z:*var. ~(z=x) /\ ~(z=y)))") ;;
```

Where `rINTER`, `SKIP` and `a_Pj` are the HOL definitions for the synchronisation operator $\sqcap$, the skip action from (6.5), and action-level projection from definition 6.3 respectively. Their precise HOL definitions are given below:

HOL-definition 7.3

```
|- !a b. a rINTER b = (\s t. a s t /\ b s t)
|- SKIP = (\s t. s = t)
|- !a A. a a_Pj A = (\s t. a (s Pj A) (t Pj A))
```

◀

So, the assignment $x, y := x + 1, y + 2$, which assigns values to two variables at the same time can now be represented by:

```
Assig2n (x,y) ((\s. (s x) + 1), (\s. (s y) + 2)).
```

7.3 UNITY programs

The quadruple (A, J, V_r, V_w) representing a UNITY program is represented by the product-type:

```
(^Action) set # ^Pred # *var set # *var set
```

However, as HOL is nimbler with predicates than with sets it has been decided to represent sets with predicates. So, instead, UNITY programs —or, to be more precise: objects of type Uprog— are represented as:

```
let Uprog = ":(^Action -> bool) # ^Pred #
              (*var -> bool)    # (*var -> bool)"
```

The destructors **a**, **ini**, **r**, and **w** used to access the components of an Uprog object are called `PROG`, `INIT`, `READ`, and `WRITE` in HOL.

HOL-definition 7.4

```
|- !P In R W. PROG(P,In,R,W) = P
|- !P In R W. INIT(P,In,R,W) = In
|- !P In R W. READ(P,In,R,W) = R
|- !P In R W. WRITE(P,In,R,W) = W
```

◀

```
 1 let Sort = new_definition
 2   (`Sort`,
 3    "Sort Labels V vOrd pOrd =
 4      (CHF{Assign2 (V i, V j) (Min vOrd (V i) (V j), Max vOrd (V i) (V j))
 5          | (i IN Labels) /\ (j IN Labels) /\ (pOrd i j)})
 6      ,
 7       TT
 8      ,
 9       CHF{V i | i IN Labels}
10      ,
11       CHF{V i | i IN Labels}"
12   ) ;;
```

Figure 12: The HOL definition of the program Sort.

The parallel composition $\|$ is called `PAR` in HOL:

HOL-definition 7.5

```
|- !Pr Qr.
    Pr PAR Qr = (PROG Pr) OR (PROG Qr),(INIT Pr) AND (INIT Qr),
                (READ Pr) OR (READ Qr),(WRITE Pr) OR (WRITE Qr)
```

To illustrate the representation of a UNITY program in HOL, consider again the program in figure 10. In applying this program to the network in Figure 9 the universe of values `*val` will be the natural numbers. But notice that in order to sort the network, `*val` can be any set of values on which a total order is defined. The code in Figure 12 is the HOL representation of this program. The code defines the constant Sort that has four parameters: the set `Labels` that contains the labels of the nodes in the network; the function `V` which given a label returns the data value that resides at the node with that label; the orderings $\prec_v$ (vOrd) and $\prec_p$ (pOrd) that define orderings on the data values and labels respectively. Note that these parameters are kept implicit in the hand definition in Figure 10. Lines 4 and 5 define the predicate which represents the set of actions of which the program Sort consists. The constant function CHF, which is defined in the library theory sets[Mel90], is used to convert a set to a predicate which given an element returns true if this element is in the set, and false otherwise. Line 7 specifies the initial condition, which is true. And finally, lines 9 and 11 are the predicates which represent the sets of read and write variables of the program respectively.

7.4 Program properties

7.4.1 Well-formedness

Definition 6.9 defined the predicate Unity to characterise all well-formed UNITY programs. The definition is re-displayed below:

$$\begin{array}{rcl} \mathsf{Unity}.P & = & (\forall a : a \in \mathbf{a}P : \Box_{\mathsf{En}} a) \ \wedge \ (\mathbf{w}P \subseteq \mathbf{r}P) \ \wedge \\ & & (\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a) \ \wedge \ (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\rightarrow a) \end{array}$$

The HOL definitions of always-enabledness, ignorance and invisibility are given below, followed by the HOL definition of the predicate Unity.

HOL-definition 7.6

```
|- !A. ALWAYS_ENABLED A = (!s. ?t. A s t)
|- !V A. V IG_BY A = (!s t. A s t ==> (s Pj V = t Pj V))
|- !V A. V INVI A =
        (!s t s' t'.
           (s Pj (NOT V) = s' Pj (NOT V)) /\ (t Pj (NOT V) = t' Pj (NOT V)) /\
           (s' Pj V = t' Pj V) /\ A s t
            ==>
           A s' t')
```

◀

Now, the HOL definition of the predicate Unity:

HOL-definition 7.7

```
|- !P In R W. UNITY(P,In,R,W) =  (!A :: P. ALWAYS_ENABLED A) /\
                                 (!A :: P. (NOT W) IG_BY A) /\
                                 (!x. W x ==> R x) /\
                                 (!A :: P. (NOT R) INVI A)
```

◀

For example, the program shown in Figure 12 can be shown to satisfy the predicate UNITY above.

7.4.2 Safety, progress and self-stabilising properties

In section 6.3 primitive operators for expressing safety, progress and self-stabilising properties of UNITY programs were discussed. The main operators were: the unless operator which is used to reason about safety properties; the stable operator to describe stable properties; the ensures operator that defines one-step progress properties; the reach ($\rightarrowtail$) operator that describes general progress; the convergence ($\rightsquigarrow$) operator to express stabilisation. Below the HOL definitions for hoare triples, unless , stable and ensures are given.

HOL-definition 7.8

```
1 |- !p A q. HOA(p,A,q) = (!s t. p s /\ A s t ==> q t)
2 |- !Pr p q. UNLESS Pr p q = (!A :: PROG Pr. HOA(p AND (NOT q),A,p OR q))
3 |- !Pr p. STABLE Pr p = UNLESS Pr p FF
4 |- !Pr p q. ENSURES Pr p q = UNITY Pr /\
5                              UNLESS Pr p q /\ (?A :: PROG Pr. HOA(p AND (NOT q),A,q))
```

◀

Compare them with their hand definition[14]:

i. $\{p\}\ a\ \{q\} \ = \ (\forall s,t :: p.s \wedge a.s.t \Rightarrow q.t)$

ii. ${}_P\vdash p$ unless $q \ = \ (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$

iii. ${}_P\vdash \circlearrowleft p \ = \ {}_P\vdash p$ unless false

[14]Notice that in the hand definition ${}_P\vdash p$ ensures q does not explicitly require that P is a UNITY program. This definition implicitly assumes that UNITY programs are considered. This assumption is not crucial for safety laws, but it is for some progress laws. In HOL, however, implicit assumptions can not be made, so it will have to be made explicit. Prasetya chose to resolve that by putting it in the definition of ENSURES.

iv. $_P\vdash p \;\mathsf{ensures}\; q \;=\; ({_P\vdash}\, p \;\mathsf{unless}\; q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\; a\; \{q\})$

In illustration, consider once again the program from figure 12. A property of this program is that during the execution of the program, the function V remains a permutation of the initial value of V (i.e. the value of V with which the program starts). Let V' denote the initial value of V, then this property can be expressed in hand notation as:

$$\vdash \circlearrowright \exists f.(\forall i : i \in \mathsf{Labels}\colon (f i) \in \mathsf{Labels}) \wedge (f \text{ is a bijection}) \wedge (\forall i : i \in \mathsf{Labels}\colon (V i) = (V'(f i)))$$

In HOL this would be:

```
STABLE  (Sort Labels V vOrd pOrd)
        (?f. (!i:: (\i. i IN Labels). (f i) IN Labels)
             /\ (BIJECTION f) /\ (!i::(\i. i IN P). V i = V' (f i)))
```

Consider again the definition of the $\mapsto$ (leads-to) operator (6.24). It is defined as the smallest transitive and disjunctive closure of ensures. Let $\mathsf{Trans}.R$ means that R is a transitive relation and Ldisj means that R is disjunctive with respect to its left argument (i.e. for all q and all non-empty sets W $(\forall p : p \in W : R.p.q) \;\Rightarrow\; R.(\exists p : p \in W : p).q$ holds (cf. the 3th item in definition 6.24)). Let TDC be defined as follows:

$$\mathsf{TDC}.R.p.q \;=\; (\forall S : R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S : S.p.q)$$

So, $\mathsf{TDC}.R$ is the smallest closure of R which is transitive and left-disjunctive, and $\mapsto$ can be defined as $(\lambda p, q.\; {_P\vdash}\, p \mapsto q) \;=\; \mathsf{TDC}.(\lambda p, q.\; {_P\vdash}\, p \;\mathsf{ensures}\; q)$. Introducing TDC is not only adding flavour to the notation, many useful properties of an operator thus defined are actually pure properties of TDC. Consequently, it would be desirable if the new $\rightarrowtail$ operator could also be defined with TDC. Now let $\mathfrak{ensures}$ be defined as follows:

$$J\; {_P\vdash}\, p \;\mathfrak{ensures}\; q \;=\; p, q \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; ({_P\vdash} \circlearrowright J) \wedge ({_P\vdash}\; J \wedge p \;\mathsf{ensures}\; q)$$

Compare this with the first item of definition 6.29. The progress operator $\rightarrowtail$ can now be defined as the TDC of $\mathfrak{ensures}$:

$$(\lambda p, q.\; J\; {_P\vdash}\, p \rightarrowtail q) \;=\; \mathsf{TDC}.(\lambda p, q.\; J\; {_P\vdash}\, p \;\mathfrak{ensures}\; q)$$

The HOL definitions are as follows:

HOL-definition 7.9

```
|- !r s. r SUBREL s = (!x y. r x y ==> s x y)
|- !r. TRANS r = (!x y z. r x y /\ r y z ==> r x z)
|- !U. LDISJ U =  (!W y. (?x. W x) /\ (!x::W. U x y) ==> U (??x::W. x) y)
|- !U x y. TDC U x y =  (!X. (SUBREL U X) /\ (TRANS X) /\ (LDISJ X) ==> X x y)
|- !Pr J p q.
    B_ENS Pr J p q =
          ENSURES Pr(p AND J)q /\ STABLE Pr J /\ (WRITE Pr) CONF p /\ (WRITE Pr) CONF q)
|- !Pr J. REACH Pr J = TDC(B_ENS Pr J)
```

◀

Finally the HOL definition of the convergence operator is:

HOL-definition 7.10

```
CON:
  |- !Pr J p q.
     CON Pr J p q =
     (WRITE Pr) CONF q /\
     (?q'. REACH Pr J p(q' AND q) /\ STABLE Pr(q' AND (q AND J)))
```

◀

8 How to design and verify UNITY programs with HOL

This section discusses how the apparatus, set out in sections 5 up to and including 7 can be used to formally design and mechanically verify a UNITY program. For those who skipped some or all of these sections, first a recapitulation is given. Section 5 describes HOL [GM93], an interactive theorem proving environment for classical higher-order logic. Section 6 deals with UNITY, a programming logic invented to support the design and verification of distributed programs. It outlines the basic foundations of UNITY (from [CM88]), and several extensions regarding compositionality and convergence (as reported in [Pra95]). Finally section 7, considers an embedding of the programming logic UNITY within HOL, which makes it possible to represent and specify UNITY programs inside the theorem prover.

The process of formally designing programs can roughly be described in seven phases. The first four phases regard the formal specification and construction of the program, the remaining phases are concerned with the proofs that are carried out in HOL.

Informally describe what the program is to do. That is write an informal specification, which reflects what the program is required to do. These informal specifications are usually written in natural language. For software companies, which write software pursuant to the wishes of their customers, this is a very important stage of developing software.

During this phase, the company must try to obtain a clear idea of what its customers want, need and expect. This is, however, not the only point in which the client must play a role. Since, especially in this informal phase, misinterpretations of the customers requirements can arise, constant interaction between the software developers and their clients is necessary throughout the whole process of software design.

Create a formal specification of what the program is to do (i.e. formally specify which problem is to be solved).

A *specification* must focus on the task and not on its eventual implementation, in other words it must specify *what* is to be done rather than *how*. A *formal specification* is a specification that is written in some specification language or logic, which has a sound mathematical basis. Writing a formal specification is central in applying formal methods to the development of programs. Formal specifications help to crystallise vague ideas, to reveal ambiguities and to expose incompleteness introduced in the requirements in the previous phase. They serve as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier and an implementer. There are many[15] specification languages or logics, which differ most in their choice of semantic domain. The best notation to use, is the one which relates most to the characteristics of the specific product being developed and the background of the individuals involved. In this report (following [Pra95]) the UNITY logic is chosen, for we want to design distributed programs, and UNITY is explicitly developed for these purposes. Furthermore it is simple and has a high level of abstraction.

Refine and decompose the formal specification. Much of program development in the UNITY methodology [CM88] consists of refining specifications (i.e. adding detail to them) and decomposing specifications (i.e. splitting them up into smaller and preferably simpler specifications). Refinement and decomposition commences by proposing a general solution strategy, by means of which the program solves the specified problem. Then the formal specification is refined according to this opposed solution strategy. Finally, the refined specification is decomposed into a set of smaller specifications. Decomposition of the specification must continue until the progress parts of the specification are solely expressed in terms of ensures . The reason for this is that ensures describes one-step progress, which – with a view to the next step – makes it easier to construct a program from the refined decomposed specification. Sundry basic laws to refine and decompose formulae of the UNITY logic were presented in section 6.

Construct a UNITY program which satisfies this refined specification. The idea in this step is that a UNITY program is formed, for which it is provable that is satisfies the refined and decomposed specification. This may however be quite difficult, since refinement and decomposition can result in a myriad specifications. Nevertheless, refined specifications usually give a clear hint as to what kind of actions should or should not be in the program, since the opposed solution strategy added some detail to how the specified problem could be solved. Moreover, decomposition is often motivated by some ideas related to the implementation of the resulting program. For example, in distributed environments designers may extensively exploit the compositionality laws, so that they will have a separate specification for each part of the program instead of a large set of specifications for the complete program.

Once a UNITY program (i.e. a program which satisfies the predicate Unity from definition 6.9) has been constructed, it is recommendable to make a pencil-and-paper proof of the program's satisfiability to the specification, despite the fact that machine-checked verification will be done in the next steps. A common mistake is to think that since mechanical verification is done later, precision and formal proofs are not required at this stage because all mistakes shall be filtered out during mechanical validation. Although the latter is partially true, such an attitude can cost lots of time and effort when a theorem prover is used. Discovering mistakes during verification in HOL is marvellous, for it demonstrates the necessity of using such a mechanical theorem prover. On the other hand, however, encountering such mistakes require that HOL definitions have to be changed and that proofs must be redone accordingly, a tedious and time consuming process which sometimes could have been prevented by more accuracy during this current stage. So summing up the motto is: "Do not get sloppy during this stage just because mechanical verification will be done in later stages, it

[15]There exists, for instance also an embedding of Z in HOL [].

can save you lots of time". Another advantage of doing formal pencil-and-paper proof during this phase, is that these proofs can considerably help in doing HOL proofs. To prove a complicated theorem in HOL, you must have some sort of proof strategy in mind with which the theorem can be proved. A pencil-and-paper proof can serve as this proof strategy, and therefore facilitates constructing a HOL proof.

This ends the formal specification and construction of a program in UNITY, the next four steps proof the correctness of the program with the aid of HOL.

Represent the program in the HOL embedding of UNITY. This also includes determination of how each component of the program is represented. For example, if arrays are used in the program then the representation of these arrays influences the ease with which certain manipulation can be carried out.

Proof that the program is well-formed. To prove the well-formedness of a program (i.e. prove that it satisfies the predicate Unity), four conditions have to be checked:

1. each action is always enabled.
2. the declared write variables should also be declared as read variables.
3. no variable not declared as a write variable is written by the program
4. no variable not declared as a read variable can influence the program.

The first two conditions are easy to proof. The third condition requires that the variables occurring at the left-hand sides of the assignments are collected and compared with the declared set of write variables. The fourth condition turns out to be [Pra95] very difficult to prove if a shallow embedding of the programming logic is not available. In the case study in section 9 the UNITY embedding will therefore be made deeper than the one presented in [Pra95], and a tactic shall be constructed which automatically proves this fourth condition for an arbitrary UNITY program.

Proof that the program satisfies the specification. First the specification must be formalised in HOL. Second, a proof tree must be constructed according to the refinement and decomposition method from the second step. Closing this proof tree constitutes of proving that the program satisfies this refined and decomposed specification

Observe that designing a program shall never proceed by consecutively working through step one up to and including seven. Program development is by no means a straight-forward one-pass process, but rather an iterative and non-linear process. A developer cannot make claims to having determined all of the requirements just because the third stage in the development process has been reached. Moreover, such claims should be considered dubious even during post-implementation phases. Thus more than once, it will be inescapable that one has to revise previous steps. For instance, because one got an additional idea, because one forgot something or simply because the customer says so. It is obvious, though very important, that when one, say for example, is in step i and wants to go back to step j to change or add something, one also has to adjust the steps in between i and j.

9 Case study

In [Pra95] a formal and mechanically verified design of Lentfert's FSA algorithm is presented, following the process described in section 8. Since the aim of this present report is to highlight the contribution to the field of formal methods and mechanical verification, this case study tackles a much simpler problem. Due to this separation of method's complexity and problem complexity, close attention can be paid to the process of formally developing a self-stabilising program without serious distraction from difficult properties of the program itself.

An instance of the sorting program, which will be studied here, was already given in section 6.4.2. The formal design of a slight generalisation of this sorting program will be illustrated in nine subsections, corresponding to the nine steps from section 8. Once again it must be stressed that designing a program does not mean consecutively working through these steps. As with most stepwise development methods it shall be necessary to revise previous steps, and work through all the steps once again. When studying the seven steps in which we designe and verify the sorting algorithm, the reader should keep this in mind.

Before the whole design process will be discussed, first some general concepts will be formally defined in the next subsection.

9.1 Some notational conventions and necessary definitions

The reason for exactly defining general concepts of which it can be assumed that everybody has a *notion* of what they are is to obtain unequivocalness. For experience shows that minor differences between notions that people have about formal concepts can cause major confusion. Moreover, when automated theorem provers are used, absolute accuracy is mandatory.

9.1.1 Notation

Sometimes $\forall x, y : x, y \in A : Q$ and $\exists x, y : x, y \in A : Q$ are abbreviated by $\forall x, y \in A : Q$ and $\exists x, y \in A : Q$ respectively. Furthermore, for any set S, $|S|$ is used to denote the cardinality of S (i.e. the number of elements in the set S). In writing an expression of the form $p \Rightarrow (q \Rightarrow r)$ we sometimes leave out the brackets.

9.1.2 Relations and orderings

Given sets A and B, a **binary relation** R from A to B is any subset of the cartesian product $A \times B$. Where $A \times B$ is short-hand for $\{(x,y)|x \in A \wedge y \in B\}$. If $(x,y) \in R$, sometimes $x\,R\,y$ is written. A binary relation on a set A is a binary relation from A to A.

Some fundamental definitions about relations and orderings are given in Figure 13.

9.1.3 Graphs

For modelling the network structure graph theoretic concepts are needed. A graph G is modelled by a pair of sets (V, E), where the set V is called the vertex set of G and set E is called the edge set of graph G. There are two kinds of graphs: directed and undirected ones.

Let R be a binary relation on set A, i.e. $R \subseteq A \times A$.

Definition 9.1

- R is **reflexive** iff $\forall x \in A \,:\, x\,R\,x$ REFL_REL
- R is **anti-reflexive** iff $\forall x \in A \,:\, \neg(x\,R\,x)$ ANTI_REFL_REL
- R is **symmetric** iff $\forall x, y \in A \,:\, (x\,R\,y) \wedge (y\,R\,x)$ SYM_REL
- R is **anti-symmetric** iff $\forall x, y \in A \,:\, \neg(x = y) \Rightarrow \neg((x\,R\,y) \wedge (y\,R\,x))$ ANTI_SYM_REL
- R is **transitive** iff $\forall x, y, z \in A \,:\, ((x\,R\,y) \wedge (y\,R\,z)) \Rightarrow (x\,R\,z)$ TRANSITIVE_REL

Definition 9.2 Partial Ordering *PARTIAL_ORDER_DEF*
R is a **partial order** iff R is reflexive, anti-symmetric and transitive.

Definition 9.3 Total Ordering *TOTAL_ORDER_DEF*
R is a **total order** iff R is a partial order and $\forall x, y \in A \,:\, (x\,R\,y) \vee (y\,R\,x)$ holds.

Definition 9.4 Transitive reflexive closure *Transitive_Refl_Closure*
The **transitive reflexive closure** of R is denoted by R^{tr}, and is defined as follows:

$$(x, y) \in R^{tr} \Leftrightarrow \exists n \in N_0 \,:\, \exists z_0, z_1, \ldots, z_n \in A \,:\, (x = z_0) \wedge (z_0\,R\,z_1\,R \ldots z_{n-1}\,R\,z_n) \wedge (z_n = y)$$

where N_0 denotes $\{0, 1, 2, \ldots\}$.

Definition 9.5 Symmetric closure *Symmetric_Closure*
The **symmetric closure** of R is denoted by R^{sym}, and is defined as follows:

$$R^{sym} = R \cup \{(x, y) | (y, x) \in R\}$$

Figure 13: Properties of relations.

◀

Definition 9.6 Directed Graph *Graph_DEF*
$G = (V, E)$ is an **directed graph** if and only if

- the vertex set V is a finite, non-empty set of vertices
- the edge set E is a binary relation over V.

◀

In directed graphs $G = (V, E)$ the edges have direction. We say that $(u, v) \in E$ is an *outgoing* edge of vertex u and an *incoming* edge of vertex v.

To each binary relation $\prec$ over a set V corresponds a directed graph:

$$G_{\prec} = (V, \{\, (u, v) \mid u \prec v \,\}).$$

Definition 9.7 UNDIRECTED GRAPH *Graph_DEF*
$G = (V, E)$ is an **undirected graph** if and only if

- the vertex set V is a finite, non-empty set of vertices
- the edge set E is a set of unordered pairs of vertices, in which self-loops are forbidden. That is
  - $E = E^{sym}$
  - $(u, u) \notin E$.

◀

9.1.4 Functions

Finally, some definitions about functions are needed. A function f that assigns to each element of a set S, a unique element in a set T is denoted by $f \in S \to T$. The set S is called the **domain** of f, denoted by $\mathsf{Dom}(f)$; the set T is called the **co-domain** of f, denoted by $\mathsf{CoDom}(f)$. For all $x \in \mathsf{Dom}(f)$, $f\,x$ is called the image of x under f. The set of all images $f\,x$ is a subset of T called the **image** of f and is denoted by $\mathsf{Im}(f)$ or $f(S)$.

Definition 9.8 INJECTIVE FUNCTION
A function $f \in S \to T$ is **injective** if and only if:
$\forall x, y \in S : (f\,x = f\,y) \Rightarrow (x = y)$.

Definition 9.9 SURJECTIVE FUNCTION
A function $f \in S \to T$ is **surjective** if and only if:
$\forall y \in T : (\exists x \in S : f\,x = y)$.
In other words, $f : S \to T$ is surjective if and only if $T = \mathsf{Im}(f)$.

Definition 9.10 BIJECTIVE FUNCTION
A function $f \in S \to T$ is **bijective** if and only if: f is both injective and surjective.

Definition 9.11 EQUAL FUNCTIONS
Two functions $f \in S \to T$ and $g \in S \to T$ are **equal**, denoted by $f = g$ if and only if:
$\forall x \in S : (f\,x) = (g\,x)$.

◀

9.2 Informally describe what the program is to do.

Our final goal is to design a self-stabilising program which sorts a network of processes in an unstable environment. To be more precise, we presume a network of processes in which:

1. every process has a unique label which is used to identify its address
2. every process has a unique local variable taken from the universe Var of all program variables (see section 6.1). These local variables can store a data value taken from the universe Val

3. some process pairs are connected via links, over which two processes can exchange data values. These links provide the only way in which two processes can communicate with each other. Communication over these connections is restricted, in that a process can only communicate with one other process at the same time.

Since we assume an unstable environment, processes and connections between them can disappear and re-appear, and external agents can tamper with the data values of the processes. Consequently, we want to design a program that does not alter the multi-set of the processes's values, and (according to some predefined orderings, $\prec_p$ and $\prec_v$, on the labels and the data values respectively) will bring the network in a state which, for any pair of processes, satisfies that the ordering ($\prec_p$) on the labels of these processes is reflected in the ordering ($\prec_v$) on the data values that reside at these processes.

9.3 Create a formal specification of what the program is to do

This subsection will start with a formal model of the network of processes in an unstable environment. After that a formal specification of the program will be constructed in terms of the convergence-operator.
A network of processes is modelled by a triple (P, C, V), where

- P is a set containing the labels of all processes in the network. Consequently, since every process has a unique label this implies that $|P|$ equals the number of processes in the network.
- C is the set of present connections between the processes, and connections are modelled by a tuple of process labels. Consequently, $C \subseteq P \times P$.
- (P, C) is considered to be an undirected graph, so according to definition 9.7 there is at least one process, the number of processes is finite and $C = C^{sym}$.
- V is a function that maps a process-label to the local variable which stores the data value that resides at that process. So $V \in P \rightarrow \mathsf{Var}$. Since every process has a unique local variable the following predicate holds:

$$\forall i, j \in P : i \neq j \Rightarrow (V\ i) \neq (V\ j) \tag{9.1}$$

A triple (P, C, V) which is in compliance with the constraints above shall be denoted by $\mathsf{Network}(P, C, V)$. As already informally specified above, such a network will be called[16] sorted, when the following property is satisfied:

$$\forall i, j \in P : i \prec_p j \Rightarrow (V\ i) \prec_v (V\ j) \tag{9.2}$$

where $\prec_p$ and $\prec_v$ are orderings on the labels and the data values respectively.

Let us start with the assumption that $\prec_v$ is a total order and determine which properties the ordering $\prec_p$ must satisfy under this assumption. Ordering $\prec_p$ has to satisfy three properties. In the first place it must be anti-symmetric. This can be illustrated with an

[16]Be aware of the overloading, which was mentioned in section 6.1, on the symbol $\prec_v$

example. Consider a network in which every process contains a different data value, that is $\forall i,j \in \mathsf{Labels} : (i \neq j) \Rightarrow ((V\ i) \neq (V\ j))$. Suppose that $\prec_p$ is not anti-symmetric, so it is possible that there are labels i and j $(i \neq j)$ for which $i \prec_p j$ and $j \prec_p i$ are both valid. Since every process contains a different value, it can be concluded from $(i \neq j)$ that $(V\ i) \neq (V\ j)$. From this and the anti-symmetric property of $\prec_v$ it can be derived that $(V\ i) \prec_v (V\ j)$ and $(V\ j) \prec_v (V\ i)$ cannot both hold at the same time. But because $i \prec_p j$ and $j \prec_p i$ both hold, the sorted network must satisfy $(V\ i) \prec_v (V\ j)$ as well as $(V\ j) \prec_v (V\ i)$, which is unachievable. In the second place $\prec_p$ must satisfy the transitivity property, for again consider a network in which every process contains a different data value. Suppose that $\prec_p$ is not transitive, so it is possible that there are i, j and k $(i \neq j \neq k)$ such that $i \prec_p j$ and $j \prec_p k$ and $k \prec_p i$ simultaneously hold. Again it can be concluded from the fact that every process contains a different value, and the anti-symmetry property of $\prec_v$, that it is impossible to sort the network. Whether $\prec_p$ is reflexive, anti-reflexive, or neither makes no difference. Because if $i \prec_p i$ holds, (9.2) is valid since $\prec_v$ is reflexive; and if $\neg(i \prec_p i)$ holds then 9.2 is trivially valid. Finally, we assume that $\prec_p$ is non-empty. Although reasons for this assumption will become apparent in later stages, it also can easily be justified at this stage, since if $\prec_p = \emptyset$ then 9.2 is a tautology and consequently there is no use in constructing a sorting program. Note that since reasons for this assumption shall be given in a later phase, obviously the need for it became clear in a later phase. Consequently, this is typically a result of program development not being a straightforward, one-pass process but rather an iterative and non-linear process.

Now a formal definition of a sorted network can be given. For this definition and the ones in the rest of this section, we shall adapt the convention that the subscript under the definition's name denotes the arguments of the definition.

Definition 9.12 SORTED NETWORK *Sorted_DEF*
Let (P, C, V) be a network. Let $\prec_p$ be a non-empty, anti-symmetric, transitive ordering on the processes' labels, and let $\prec_v$ be a total ordering on the data values of the processes. This network is defined to be sorted if it satisfies:

$$\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)} = \forall i,j \in P : i \prec_p j \Rightarrow (V\ i) \prec_v (V\ j)$$

◀

By contrast, compare definition 9.12 with the informal description that was given in the previous section. It is clear that the former is less confusing than the latter.

Until now, we have made the following assumptions about the environment in which the program will operate. First, we assumed an unstable environment in which external agents can cause arbitrary transient errors. Second, we assumed a network of processes, in which two processes can only compare their data values if they have a connection between them, and a process can only compare its value with one other value at the same time. In order to sort all the data values of a network, a sufficient number of these values have to be compared (i.e. a sufficient number of connections must be present) and appropriate actions must be taken according to the result of this comparison. Of course the way these values are compared and which actions will be taken accordingly are matters of *how* the program will achieve *what* it is to do, and must not be part of the specification. But conditions on the environment in which the program is required to operate are matters of *what* the program

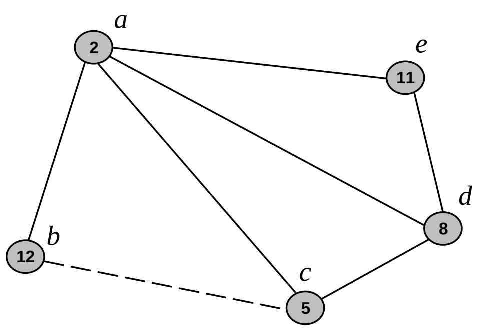

Figure 14: A network which cannot be sorted.

is to do, viz. under what circumstances it must fulfill its requirements. Although, at this stage, we may not make assumptions whatsoever on how the program will achieve the requested results, nevertheless we have to take full account of the limitations the environment imposes upon the possible implementations of the program, by specifying and if necessary strengthening the properties of this environment[17]. It is obvious that, in this case, we have to curtail the set of possible failures that can change the environment, and settle for convergence instead of self-stabilisation. As it happens, we cannot allow arbitrary connections to disappear. Consider for example the network in Figure 14, where the labels (written above the processes) are characters, and the data values (written inside the processes) are numbers. As a result of an environmental failure, the connection between processes b and c has disappeared. Let $\prec_p$ and $\prec_v$ be the lexicographic order on characters and the less-than-or-equal ($\leq$) order on numbers respectively. There exists no implementation whatsoever which does not alter the multi-set of the processes's values and can sort this network, since:

- the values of processes a, c, d and e are already sorted according to definition 9.12, so these processes shall not undertake any action[18]
- the same holds for the processes a and b
- process b cannot compare its value with any of processes c, d and e, so nothing will be done by process b either.

As exemplified above, we cannot allow arbitrary failures of connections. As a consequence we must formalise a condition which states when there are still enough connections left for any implementation to sort the network. This condition then imposes a restriction on the set of failures from which the convergent program can recover. A minimal and sufficient

[17]It can occur that the limitations the environment enforces upon the possible implementations do not become clear till one reaches the implementation phase. In this case one has to return to the specification phase and modify the latter.

[18]Note that by concluding that no action will be undertaken, we do *not* make assumptions about the eventual implementation. We merely use the fact that a convergent program is being designed, which means that if the program finds itself in the required situation it will stay there.

condition on the extant connections must imply that if the network is not yet sorted, then there always must be processes which "know" that their values are not sorted; in other words, among the pairs of connected processes whose labels are ordered by $\prec_p$, there must at least be one pair whose values are out-of-order. For, if this condition is satisfied, it will always be possible to undertake some action if the network is not yet sorted. Consequently, situations as sketched in the example above cannot arise. Before this condition is formalised, first some definitions and theorems[19] are given.

Definition 9.13 WRONG PAIRS *Wrong_Pairs_DEF*

$$\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)} = \{(i,j) | i \in P \wedge j \in P \wedge (i \prec_p j) \wedge \neg((V\ i) \prec_v (V\ j))\}$$

Definition 9.14 WRONG PAIR *Wrong_Pair_DEF*

$$\mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)} = (i,j) \in \mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}$$

Theorem 9.15 *Nr_Of_Wrong_Pairs_GREATER_0_IMP_EXISTS_WP*

$$\neg\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)} \Leftrightarrow \exists i,j \in P : \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)} \Leftrightarrow |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| > 0$$

Theorem 9.16 *WRONG_PAIRS_EMPTY_EQ__SORTED, Nr_Of_WrongPairs_0_EQ_WrongPairs_EMPTY*

$$\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)} \Leftrightarrow (\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)} = \{\}) \Leftrightarrow |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0$$

◀

Thus the condition, from now on called Cond, which we are looking for must satisfy:

$$\mathsf{Cond} \Rightarrow \neg\,\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)} \Rightarrow \exists u,v \in P : (u,v) \in C^{sym} \wedge u \prec_p v : \mathsf{WrongPair}^{(u,v)}_{(P,C,V,\prec_p,\prec_v)} \quad (9.3)$$

Where C^{sym} is used to model the bi-directional property of the connections in the network. Now, we state that the following definition of Cond satisfies 9.3, the proof of which shall be given below.

$$\mathsf{Cond} = (\prec_p \subseteq (\prec_p \cap (C^{sym}))^{tr}) \quad (9.4)$$

In order to prove that definition 9.4 of Cond satisfies 9.3, it suffices to prove that, under the assumptions

ass1 $(\prec_p \subseteq (\prec_p \cap (C^{sym}))^{tr})$

ass2 $\neg\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}$

it holds that $(\exists u,v \in P : (u,v) \in C^{sym} \wedge u \prec_p v : \mathsf{WrongPair}^{(u,v)}_{(P,C,V,\prec_p,\prec_v)})$.

Assumption **ass2** imparts that there exist a pair of processes, say i and j, of which the values are not sorted, i.e. $i \prec_p j$ and $\neg(V\ i \prec_v V\ j)$ hold at the same time. Consequently, assumption **ass1** tells us that $(i,j) \in (\prec_p \cap (C^{sym}))^{tr}$. Then by expanding definition 9.4 the following can be derived:

[19]The proofs of the theorems are not given, for they follow trivially from the definitions 9.12, 9.14 and 9.13.

$$\begin{array}{l}\exists n \in N_0 \ : \\ \quad (\exists z_0, z_1, \ldots, z_n \in P \ : \\ \quad (i = z_0) \wedge (z_0 \ \prec_p \ z_1 \ \prec_p \ldots z_{n-1} \ \prec_p \ z_n) \wedge (z_n = j)) \wedge \neg((V \ z_0) \prec_v (V \ z_n)) \\ \quad \wedge \\ \quad (\forall k \ : 0 \leq k < n \ : (z_k, z_{k+1}) \in (C^{sym}))\end{array} \tag{9.5}$$

Instantiate 9.5 with $n \in N_0$; and instantiate the left conjunct of the resulting predicate with $z_0, z_1, \ldots, z_n \in P$. Then it can be proved that

$$\exists k \ : 0 \leq k < n \ : \neg(V \ z_k) \prec_v (V \ z_{k+1}) \tag{9.6}$$

For, suppose 9.6 does not hold (i.e. $\forall k \ : 0 \leq k < n \ : (V \ z_k) \prec_v (V \ z_{k+1})$), then we can conclude, from the transitivity of $\prec_v$, that $((V \ z_0) \prec_v (V \ z_n))$, which evidently contradicts with the instantiation of 9.5. From 9.5 and 9.6 it can be easily derived that

$$\exists k \ : 0 \leq k < n \wedge (z_k, z_{k+1}) \in C^{sym} \wedge z_k \prec_p z_{k+1} \ : \mathsf{WrongPair}^{(z_k, z_{k+1})}_{(P,C,V,\prec_p,\prec_v)}) \tag{9.7}$$

which trivially establishes the proof that:

$$(\exists u, v \in P : (u, v) \in C^{sym} \wedge u \prec_p v : \mathsf{WrongPair}^{(u,v)}_{(P,C,V,\prec_p,\prec_v)})$$

.

Thus, the minimal and sufficient condition on the extant connections can be formally defined by the following definition.

Definition 9.17 *Sufficient_Connection_In_Network_For_Sorting*
Let (P, C, V) be a network. Let $\prec_p$ be an ordering on the processes' labels.

$$\mathsf{SufficientConnections}_{(C,\prec_p)} = (\prec_p \subseteq (\prec_p \cap (C^{sym}))^{tr})$$

◀

And the theorem which follows from 9.3 , 9.4 and the proof sketched above.

Theorem 9.18 *Nr_of_Wrong_Pairs_GREATER_0_IMP_EXISTS_CONNECTED_WP*
For any network of processes $\mathsf{Network}(P, V, C)$, total ordering $\prec_v$ on the processes' data values, and non-empty, anti-symmetric and transitive ordering $\prec_p$ on the processes' labels:

$$\frac{\mathsf{SufficientConnections}_{(C,\prec_p)} \wedge (\neg\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)})}{\exists u, v \in P : (u, v) \in C^{sym} \wedge u \prec_p v : \mathsf{WrongPair}^{(u,v)}_{(P,C,V,\prec_p,\prec_v)})}$$

◀

Now, we are almost ready to construct the formal specification of what the program is to do. We have defined what the program must establish, i.e. sort a particular kind of network of processes (definition 9.12), we have defined under which conditions it must achieve that, i.e. there is a total ordering on the processes' data values; there is an anti-symmetric and transitive relation on the processes' labels; and there is a restriction upon the failures that may occur (definition 9.17). There is, however, one, important but obvious, thing that must be embodied in the specification. During the activity of sorting the network, we want the distribution of the values among the processes to remain a permutation of the initial distribution (i.e. the one with which the program started). If we do not require this, a simple program which just assigns the same value to all processes would achieve, by reflexitivity of $\prec_v$, a sorted network (according to definition 9.12). And this is obviously not what we want. Evidently, the distribution of the values among the processes in a network $\mathsf{Network}(P, C, V)$, is described by the function V. So if we presume that V' is the initial value of V, we demand that, during the whole execution of the program, V is a permutation of V'. In other words, we require that $\mathsf{Permutation}_{(P,V,V')}$ (see definition 9.19 below) is an invariant of our program.

Definition 9.19 PERMUTATION *Perm_DEF*

Let P be a set of values, and let V and V' be two functions such that

- $\mathsf{Dom}(V) = \mathsf{Dom}(V') = P$
- $\mathsf{Im}(V) = \mathsf{Im}(V')$

Then V is a **permutation** of V' if:

$$\begin{array}{l}\mathsf{Permutation}_{(P,V,V')} = \\ \qquad \exists f \in P \to P : (\forall i \in P : (f\ i) \in P) \wedge (\mathsf{Bijection} f) \wedge (\forall i \in P : (V\ i) = (V'(f\ i)))\end{array}$$

---◀

We are now ready to present the formal specification of the program in terms of the convergence operator.

Specification 9.20

$$\frac{\begin{array}{c}\mathsf{Network}(P, C, V) \wedge \mathsf{Total}(\prec_v, V) \wedge (\prec_p \neq \emptyset) \\ \mathsf{AntiSymmetric}(\prec_p, P) \wedge \mathsf{Transitive}(\prec_p, P) \wedge \mathsf{SufficientConnections}_{(C,\prec_p)}\end{array}}{(\vdash \Box\mathsf{Permutation}_{(P,V,V')}) \wedge (\mathsf{Permutation}_{(P,V,V')} \vdash (V = V') \rightsquigarrow \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)})}$$

---◀

9.4 Refine and decompose the specification

This subsection describes the refinement and decomposition of convergence-part of specification 9.20. Since HOL verification is done in latter stages, the intermediate predicates, which result from refinement or decomposition by applying UNITY rules, are written down with accuracy. Consequently, some predicates may appear strange to the reader, because these predicates contain superfluous information. The validity of $\mathsf{Network}(P, C, V)$, $\mathsf{Transitive}(\prec_p)$,

$\mathsf{SufficientConnections}_{(C,\prec_p)}$, $\mathsf{AntiSymmetric}(\prec_p)$ and $\mathsf{Total}(\prec_v)$ shall be implicitly assumed. Consequently, the specification which will be refined is:

$$\mathbf{S_0} \ : \ \mathsf{Permutation}_{(P,V,V')} \vdash (V = V') \rightsquigarrow \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}$$

The solution strategy[20] which shall be used to refine $\mathbf{S_0}$, is one that reduces the number of wrong pairs of processes. In other words, during the execution of a program – which uses this strategy to sort a network (as specified by $\mathbf{S_0}$) – the number of wrong pairs of processes must be reduced. Let $|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}|$ denote the number of elements in the set $\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}$, that is the number of wrong pairs in the network. Theorem 9.16 imparts that in a sorted network there are no wrong pairs of processes (i.e. $|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0$). Consequently, since $|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}|$ is always a value from N_0, the less-than ($<$) is known to be a well-founded relation on N_0, and since the value of $|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}|$ reduces during the execution of a program that exploits our solution strategy, the network shall eventually get sorted. We shall now refine $\mathbf{S_0}$ according to this opposed solution strategy. For the sake of readability:

- all predicate confinement constraints, which must be satisfied in order for some laws to be applicable, are gathered into a set called Conf, which shall be expanded at the end of this section.
- The stability requirements (i.e. $\mathsf{Permutation}_{(P,V,V')}$ in $\mathbf{S_0}$) are omitted from the specifications. So $\mathbf{S_0}$ becomes:

$$\mathbf{S_0} \ : \ (V = V') \rightsquigarrow \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}$$

Before we continue it must be pointed out that when mechanical verification is attempted, one must be prepared to deal with every detail explicitly.

Let us start by rewriting specification $\mathbf{S_0}$ into a more suitable form. Since everything implies true and $\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)} = (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$, we can use $\rightsquigarrow$ SUBSTITUTION (theorem 6.41) to derive:

$$\mathbf{S_0} \ : \ (V = V') \rightsquigarrow \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}$$

$\Leftarrow$ ($\rightsquigarrow$ SUBSTITUTION 6.41)

$$\mathbf{S_1} \ : \ \mathsf{true} \rightsquigarrow (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$$

where, behind the scenes, the set Conf becomes:

$$\{(V = V'), \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}\}$$

Now $\mathbf{S_1}$ shall be refined according to the solution strategy informally described above. Recall the BOUNDED PROGRESS principle for $\rightsquigarrow$ (theorem 6.46).

$$P, J : \ \frac{(q \rightsquigarrow q) \ \wedge \ (\forall M :: p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q)}{p \rightsquigarrow q}$$

[20]There are other possible solution strategies, see for example [CM88].

The principle states that if A is a set of values on which a well-founded relation $\prec$ is defined and if $m \in A$, then in a UNITY program P which will either establish q or decrease m, eventually q will hold since $\prec$ is well-founded and hence cannot be decreased forever. Evidently our solution strategy is an instance of this principle. Let us apply this principle to $\mathbf{S_1}$:

$\mathbf{S_1}$: $\mathsf{true} \rightsquigarrow (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$

$\Leftarrow$ ($\rightsquigarrow$ BOUNDED PROGRESS 6.46, $<$ well-founded on N_0, $|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| \in N_0$)

$\mathbf{S_2}$: $(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0) \rightsquigarrow (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$
$\wedge$
$\mathbf{S_3}$: $\forall m \in N_0$:
$|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m \rightsquigarrow (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m \vee |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$

$\mathbf{S_2}$ can be decomposed into smaller specifications, using $\rightsquigarrow$ REFLEXIVITY and $\circlearrowleft$ CONJUNCTION:

$\mathbf{S_2}$: $(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0) \rightsquigarrow (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$

$\Leftarrow$ ($\rightsquigarrow$ REFLEXIVITY 6.42)

$\mathbf{S_{2a}}$: $\circlearrowleft \mathsf{Permutation}_{(P,V,V')}$
$\wedge$
$\mathbf{S_{2b}}$: $\circlearrowleft (\mathsf{Permutation}_{(P,V,V')} \wedge |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$

$\Leftarrow$ ($\circlearrowleft$ CONJUNCTION 6.22)

$\mathbf{S_{2c}}$: $(\circlearrowleft \mathsf{Permutation}_{(P,V,V')}) \wedge (\circlearrowleft |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$

Decomposing $\mathbf{S_3}$ is much more difficult since it requires some important observations. First, observe that for the case that $m = 0$, $\mathbf{S_3}$ boils down to $\mathbf{S_2}$. Since specifications $\mathbf{S_{2a}}$ and $\mathbf{S_{2b}}$ already cover this, we can assume $m > 0$ in decomposing $\mathbf{S_3}$. Consequently, $(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m)$ equals $(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| > 0)$, which is the same as $\neg\mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}$. As a result, the following lemma can easily be proved for all $m > 0$ using Theorem 9.18 and the implicit assumption that $\mathsf{SufficientConnections}_{(C,\prec_p)}$.

Lemma 9.21 *Nr_of_Wrong_Pairs_GREATER_0_IMP_EXISTS_CONNECTED_WP*

$$\frac{(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge m > 0}{\exists i,j \in P : (i,j) \in C^{sym} \wedge i \prec_p j : (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)})}$$

◀

The following lemma is trivially true for all m. We will need it in a moment in order to apply $\rightsquigarrow$ SUBSTITUTION (6.41) to $\mathbf{S_3}$, in order to prepare the latter for $\rightsquigarrow$ DISJUNCTION (6.44).

Lemma 9.22

$$\frac{\exists i,j \in P : (i,j) \in C^{sym} \wedge i \prec_p j : (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m)}{(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m) \vee (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)}$$

◀

Now, we can rewrite $\mathbf{S_3}$ as follows:

$$\begin{aligned}&\mathbf{S_3} : \forall m \in N_0 : \\ &\quad |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m \leadsto (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m \vee |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)\end{aligned}$$

$\Leftarrow$ ($\leadsto$ SUBSTITUTION 6.41, lemmas 9.21 and 9.22 and $m > 0$)

$$\begin{aligned}&\mathbf{S_4} : \forall m : m > 0 : \\ &\quad \exists i,j \in P : (i,j) \in C^{sym} \wedge i \prec_p j : (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)} \\ &\quad \leadsto \\ &\quad \exists i,j \in P : (i,j) \in C^{sym} \wedge i \prec_p j : |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m\end{aligned}$$

In order to be able to apply $\leadsto$ DISJUNCTION to $\mathbf{S_4}$, it must hold that:

$$\{(i,j) \mid i,j \in P \wedge (i,j) \in C^{sym} \wedge (i \prec_p j)\} \neq \{\}$$

Now one rationale for the assumption $\prec_p \neq \emptyset$, which was made on page 47, pops up, because the set $\{(i,j) \mid i,j \in P \wedge (i,j) \in C^{sym} \wedge (i \prec_p j)\}$ cannot be proved non-empty if $\prec_p$ can be empty. Since $\prec_p \neq \emptyset$ is a natural assumption, we were not reluctant to add it in order to proceed this line of refinement. Consequently, at this point we went back to phase two and added the assumption $\prec_p \neq \emptyset$, thereby endorsing that program development is an iterative and non-linear process.

The proof that $\{(i,j) \mid i,j \in P \wedge (i,j) \in C^{sym} \wedge (i \prec_p j)\}$ is not empty is given after the following lemma that formally states this proof obligation:

Lemma 9.23 *EXISTS_AT_LEAST_ONE_CONNECTION_THAT_CAN_BE_A_WP*

$$\frac{\prec_p \neq \emptyset \wedge \mathsf{SufficientConnections}_{(C,\prec_p)}}{\{(i,j) \mid i,j \in P \wedge (i,j) \in C^{sym} \wedge (i \prec_p j)\} \neq \emptyset}$$

◀

We shall prove this lemma by contradiction.

Assume $\prec_p \neq \emptyset$, $\mathsf{SufficientConnections}_{(C,\prec_p)}$ and $\{(i,j) \mid i,j \in P \wedge (i,j) \in C^{sym} \wedge (i \prec_p j)\} = \emptyset$. The last assumption implies that $(\prec_p \cap C^{sym}) = \emptyset$. From $\mathsf{SufficientConnections}_{(C,\prec_p)}$ it can then be derived that $\prec_p = \emptyset$, which contradicts the assumptions.

Now $\leadsto$ Disjunction can be applied to $\mathbf{S_4}$:

$\mathbf{S_4}$: $\forall m : m > 0 :$
$\exists i, j \in P : (i, j) \in C^{sym} \wedge i \prec_p j : (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)}$
$\leadsto$
$\exists i, j \in P : (i, j) \in C^{sym} \wedge i \prec_p j : |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m$

$\Leftarrow$ ($\leadsto$ Disjunction 6.44 and lemma 9.23)

$\mathbf{S_5}$: $\forall m : m > 0 :$
$\forall i, j \in P : (i, j) \in C^{sym} \wedge i \prec_p j :$
$((|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)}) \leadsto (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m)$

$\Leftarrow$ ($\leadsto$ Introduction 6.40)

$\mathbf{S_{6a}}$: $\forall m : m > 0 :$
$\circlearrowleft (\mathsf{Permutation}_{(P,V,V')} \wedge |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m)$
$\wedge$
$\mathbf{S_{6b}}$: $\forall m : m > 0 :$
$\forall i, j \in P : (i, j) \in C^{sym} \wedge i \prec_p j :$
$\mathsf{Permutation}_{(P,V,V')} \wedge ((|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)})$
ensures
$|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m$

The refinement is now completed since the progress parts are solely expressed in terms of ensures . So we have decomposed and refined specification $\mathbf{S_0}$ into:

$$\mathbf{S_0} \Leftarrow \mathbf{S_{2c}} \wedge \mathbf{S_{6a}} \wedge \mathbf{S_{6b}} \wedge \mathsf{Conf}$$

Which, after applying $\circlearrowleft$ Conjunction to $\mathbf{S_{6a}}$, comes down to:

$\mathbf{S_0}$: $\mathsf{Permutation}_{(P,V,V')} \vdash (V = V') \leadsto \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}$

$\Leftarrow$

$\mathbf{S_{2c}}$: $(\circlearrowleft \mathsf{Permutation}_{(P,V,V')}) \wedge (\circlearrowleft |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$
$\wedge$
$\mathbf{S_{6a}}$: $(\forall m : m > 0 : \circlearrowleft (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m))$
$\wedge$
$\mathbf{S_{6b}}$: $\forall m : m > 0 :$
$\forall i, j \in P : (i, j) \in C^{sym} \wedge i \prec_p j :$
$(\mathsf{Permutation}_{(P,V,V')}) \wedge ((|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)})$
ensures
$|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m$
$\wedge$
Conf

```
prog    Sort
read    $\{V\ i \mid i \in P\}$
write   $\{V\ i \mid i \in P\}$
init    $V = V'$
assign  $[\![ i,j : (i,j \in P) \wedge (i \prec_p j) \wedge (i,j) \in C^{sym} ::$
            $(V\ i),(V\ j) := \mathsf{min}_{\prec_v}(V\ i, V\ j),\ \mathsf{max}_{\prec_v}(V\ i, V\ j)$
```

Figure 15: The sorting program

Where Conf is the set:

$$\begin{array}{ll} \{ & (V = V'), \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}, \\ & (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0), \\ & |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m, \\ & |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m, \\ & \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)} \\ \} & \end{array}$$

9.5 Construct a program which satisfies this refined specification

Now, a UNITY program must be constructed from the refined and decomposed specification which was given at the end of the previous section. Considering the properties of the network – principally the property that a process can only communicate with one other process at the same time – it is evident that the only thing two connected processes can do is compare their values and swap them if they are out-of-order with respect to the processes' labels. The resulting program, which differs only slightly from the one given in Figure 10, is presented in Figure 15. Note that the program performs a topological sort on the directed acyclic graph $G_{\prec_p}$.

To prove that the program in Figure 15 is a well-formed UNITY program. According to section 6.2.2 we must show that the program is an element of the language generated by the grammer which was presented in section 6.2.1, and that the following four requirements are met.

i. The actions of the program should be always-enabled

ii. A write variable is also readable.

iii. The actions of a program should only write to the declared write variables.

iv. The actions of a program should only depend on the declared read variables.

The validity of these requirements can trivially be established by inspecting the program code in Figure 15.

In order to verify that the UNITY program Sort satisfies specification 9.20, we have to show that, if Network(P, C, V),SufficientConnections$_{(C,\prec_p)}$, AntiSymmetric$(\prec_p)$, Total$(\prec_v)$

and $\mathsf{Transitive}(\prec_p)$ hold, then the program satisfies $\mathbf{S_0}$ and $(\vdash \Box\mathsf{Permutation}_{(P,V,V')})$. Before we do this, let us first look more closely at the program and some of its properties.

All actions, which can be non-deterministically selected during the execution of the program, do the same: the data values of two connected processes are compared and swapped if and only if these values are out-of-order with respect to the processes's labels. Swapping the data values of two connected processes i and j, means that process i stores the data value of process j in its local variable and vice versa. Furthermore, during the execution of an action only the values that reside in wrong pairs of processes are swapped, and only two processes at the same time are considered so that the data values of all other processes are unchanged. The following definition states the predicate which holds after executing an action which swapped two data values. Since we are considering state transitions caused by singular actions, we have to abandon the overloading mentioned in section 6.1 and explicitly write down the states in this definition. While the only variables present in the program are those returned by the function V, we can simply eliminate the overloading by composing all occurrences V with the right state. And because, as a notational convention, the subscripts of the predicates (e.g. WrongPairs) contained the arguments of the predicate, replacing V by for example $(s \circ V)$ raises no problems.

Definition 9.24 Two processes have swapped their values *Two_Elts_Swapped_DEF*

For all $a \in \mathbf{a}Sort$, if $a.s.t$ holds then executing action a in state s resulted in swapping two data values of a wrong pair of processes if and only if the predicate $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$ holds.

$$\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)} =$$

$$\begin{aligned}\exists i,j \in P : \ & \mathsf{WrongPair}^{(i,j)}_{(P,(s\circ V),\prec_p,\prec_v)} \\ & \wedge (\forall k \in P : (k \neq i \wedge k \neq j) \Rightarrow (s\circ V)\,k = (t\circ V)\,k) \\ & \wedge (s\circ V)\,i = (t\circ V)\,j \wedge (s\circ V)\,j = (t\circ V)\,i\end{aligned}$$

◀

Every action swaps the data values of two connected processes only if these values are out-of-order, and does nothing otherwise. Consequently, the following can be easily inferred for all actions $a \in \mathbf{a}Sort$ and all states s and t:

$$\frac{a.s.t}{\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)} \vee (s \circ V) = (t \circ V)} \tag{9.8}$$

Now let us turn to our specification and show that our program satisfies $\mathbf{S_{2c}}$, $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$.

It can easily be verified that program *Sort* satisfies $\mathbf{S_{2c}}$. Firstly, if an action of the program results in changing a process' data value then this value is exchanged (i.e. substituted) for a data value of another process. Consequently, the distribution of the data values among the processes remains a permutation of the initial distribution, i.e. $\mathsf{Permutation}_{(P,V,V')}$ is stable. Secondly, since only out-of-order-pairs are swapped no swapping whatsoever will be done if the network is sorted, so $(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = 0)$ is stable. Less trivial, however, is to recognise that the stability predicate $\mathbf{S_{6a}}$ and the progress property $\mathbf{S_{6b}}$ are satisfied. In order to show their satisfiability, it turns out to be sufficient to prove that our program employs the solution strategy which was introduced in section 9.4, that is if two processes swap their data values, then the total number of wrong pairs of processes decreases. In other words, we must prove that *if* the program finds itself in some state s in which holds that there still exist wrong pairs of processes, *then if* t is the state in which the program results after swapping the data values of some connected wrong pair (which exists because of theorem 9.18), *then* the number of wrong pairs in state t is less than the number of wrong pairs in state s. More formally, in order to show that $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$ are satisfied by the program, it suffices to betoken that for all $m \in N_0$ and states s and t:

$$\frac{|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = m \wedge \mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}}{|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < m} \tag{9.9}$$

Let us first assume that 9.9 holds and prove that the program satisfies $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$.

$\mathbf{S_{6a}}$: $(\forall m : m > 0 : \circlearrowleft(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m))$

= (definition 6.16 and 6.13)

$\forall m : m > 0 : (\forall a \in \mathbf{a}Sort : \{|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m\}a\{|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m\})$

= (6.2)

$\forall m : m > 0 : (\forall a \in \mathbf{a}Sort : \forall s,t ::$
$(|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| < m \ \wedge\ a.s.t) \Rightarrow\ |\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < m)$

So, to show that $\mathbf{S_{6a}}$ is satisfied it must be demonstrated that for all $m > 0$, for all actions $a \in \mathbf{a}Sort$ and for all states s and t holds that:

$$\frac{|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| < m \wedge a.s.t}{|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < m}$$

Assume $|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| < m$ and $a.s.t$, then, following 9.8, there are two possible cases that can be distinguished:

Case $(s \circ V) = (t \circ V)$ is easy, since $(s\circ V) = (t\circ V)$ implies that $\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}$ equals $\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}$, which immediately establishes the proof.

Case $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$— From the assumption that $|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| < m$ we can deduce that there exists a $k < m$ such that $|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = k$. From 9.9 we then know that $|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < k$, which establishes the proof since $k < m$.

This proves the requirement that $\mathbf{S_{6a}}$ is satisfied by the program from Figure 15. Now let us turn to $\mathbf{S_{6b}}$.

$\mathbf{S_{6b}}$: $\forall m : m > 0 :$
$\forall i,j \in P : (i,j) \in C^{sym} \wedge i \prec_p j :$
$\mathsf{Permutation}_{(P,V,V')} \wedge (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)}$
ensures

$$|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m$$

= (definition of **ensures** 6.11)

$$\begin{array}{l}
\forall m : m > 0 : \\
\quad \forall i,j \in P : (i,j) \in C^{sym} \wedge i \prec_p j : \\
\quad \mathsf{Permutation}_{(P,V,V')} \wedge (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)} \\
\quad\ \mathsf{unless} \\
\quad |\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m \\
\quad \wedge \\
\quad \exists a \in \mathbf{a}Sort : \\
\quad\ \{\mathsf{Permutation}_{(P,V,V')} \wedge (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| = m) \wedge \\
\quad\quad \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)}) \wedge \neg(|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m) \\
\quad\ \}\ a\ \{|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m\}
\end{array}$$

So, to confirm that $\mathbf{S_{6b}}$ is satisfied it must be argued that for all m, for all actions $a \in \mathbf{a}Sort$, for all processes i and j and for all states s and t the following two conjectures hold:

($\mathbf{C_1}$) After rewriting with definition 6.10 and with (6.2) (the latter which can be found on 17)

$$\frac{\begin{array}{c} a.s.t\ \wedge\ m > 0\ \wedge\ (i,j) \in C^{sym}\ \wedge\ \mathsf{Permutation}_{(P,(s \circ V),V')} \\ \mathsf{WrongPair}^{(i,j)}_{(P,V,\prec_p,\prec_v)}\ \wedge\ (|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = m) \\ \neg\ (|\mathsf{WrongPairs}_{(P,V,\prec_p,\prec_v)}| < m) \end{array}}{\begin{array}{c} (\mathsf{Permutation}_{(P,(t\circ V),V')}\ \wedge\ (|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| = m) \\ \mathsf{WrongPair}^{(i,j)}_{(P,(t\circ V),\prec_p,\prec_v)})\ \vee\ |\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < m \end{array}}$$

($\mathbf{C_2}$)

$$\exists a \in \mathbf{a}Sort : \frac{\begin{array}{c} a.s.t\ \wedge\ m > 0\ \wedge\ (i,j) \in C^{sym}\ \wedge\ \mathsf{Permutation}_{(P,(s\circ V),V')} \\ \mathsf{WrongPair}^{(i,j)}_{(P,(s\circ V),\prec_p,\prec_v)}\ \wedge\ (|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = m) \\ \neg\ (|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| < m) \end{array}}{|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < m}$$

Conjecture $\mathbf{C_1}$ can be easily proved, for if, after executing action a:

- $(s \circ V) = (t \circ V)$ holds, then the first disjunct of $\mathbf{C_1}$'s conclusion trivially holds
- $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$ holds, then 9.9 immediately confirms the second disjunct of $\mathbf{C_1}$'s conclusion.

In order to prove conjecture $\mathbf{C_2}$, we must show that there exists an action which reduces the number of wrong pairs of processes, given that there exists at least one connected wrong pair of processes. Because $(i,j) \in C^{sym}$ can be assumed, this action is:

$$a = (t \circ V)\ i), (t \circ V)\ j) := \mathsf{min}_{\prec_v}((s \circ V)\ i, (s \circ V)\ j), \mathsf{max}_{\prec_v}((s \circ V)\ i, (s \circ V)\ j),$$

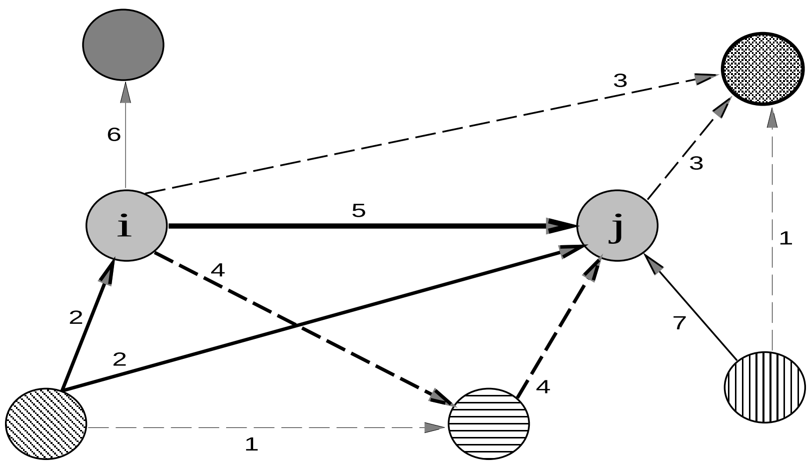

Figure 16: The possible edges in the set $\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}$, when $(i,j) \in \mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}$.

which, since $\mathsf{WrongPair}^{(i,j)}_{(P,(s\circ V),\prec_p,\prec_v)}$ can be assumed, equals

$$a = (t \circ V)\, i, (t \circ V)\, j := (s \circ V)\, j, (s \circ V)\, i).$$

Instantiating $\mathbf{C_2}$'s assumptions with this action a, validates $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$ and $(|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = m)$, which, using 9.9, establishes $\mathbf{C_2}$.

Recapitulating, we have shown that if 9.9 holds, then our program *Sort* satisfies the sub-specifications $\mathbf{S_{6a}}$ and $\mathbf{S_{6b}}$. So to finish the verification of our program's satisfiability to $\mathbf{S_0}$, it suffices to show that all elements of Conf are elements of $\mathsf{Pred}.(\mathbf{w}Sort)$, and that theorem 9.9 holds. The latter is repeated below for convenience:
for all $m \in N_0$ and states s and t:

$$\frac{|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = m \wedge \mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}}{|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < m} \tag{9.9}$$

Assume $|\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}| = m$ and $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$. From these assumptions we can infer that there exists a pair of processes of which the data values are swapped during the state-transition from s to t, let us call this pair (i,j). Now, for an arbitrary state s, we look at $\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}$ and split this set up in seven disjoint sets using i and j.

For any state s:

$$\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)} = \mathsf{WP}^1_s \cup \mathsf{WP}^2_s \cup \mathsf{WP}^3_s \cup \mathsf{WP}^4_s \cup \mathsf{WP}^5_s \cup \mathsf{WP}^6_s \cup \mathsf{WP}^7_s \tag{9.10}$$

where

$\mathsf{WP}^1_s = \{(x,y)|x,y \in P \ \wedge \ x \prec_p y \ \wedge \neg((s \circ V)x \prec_v (s \circ V)y) \ \wedge \ x \neq i \wedge x \neq j \wedge y \neq i \wedge y \neq j\}$

$\mathsf{WP}^2_s = \{(x,y)|x \in P \wedge x \neq i \ \wedge x \neq j \wedge x \prec_p i \wedge (y = i \vee y = j) \wedge \neg((s \circ V)x \prec_v (s \circ V)y)\}$

$\mathsf{WP}^3_s = \{(x,y)|y \in P \wedge y \neq i \ \wedge y \neq j \wedge j \prec_p y \wedge (x = i \vee x = j) \wedge \neg((s \circ V)x \prec_v (s \circ V)y)\}$

$\mathsf{WP}^4_s = \{(x,y)|((x = i \wedge y \neq j \wedge i \prec_p y \wedge y \prec_p j) \vee (y = j \wedge x \neq i \wedge i \prec_p x \wedge x \prec_p j))$
$\wedge \neg((s \circ V)x \prec_v (s \circ V)y)\}$

$\mathsf{WP}^5_s = \{(i,j)|i \prec_p j \ \wedge \neg((s \circ V)i \prec_v (s \circ V)j`)\}$

$\mathsf{WP}^6_s = \{(i,y)|y \neq j \wedge i \prec_p y \wedge \neg(j \prec_p y) \wedge \neg(y \prec_p j) \wedge \neg((s \circ V)i \prec_v (s \circ V)y)\}$

$\mathsf{WP}^7_s = \{(x,j)|x \neq i \wedge x \prec_p j \wedge \neg(x \prec_p i) \wedge \neg(i \prec_p x)\neg((s \circ V)x \prec_v (s \circ V)j)\}$

and:

$$\forall k,l : k,l = 1,2,\ldots,7 \wedge k \neq l : \mathsf{WP}^k_s \cap \mathsf{WP}^l_s = \emptyset \qquad (9.11)$$

From Figure 16, which shows the directed graph $G_{\prec_p}$, one can deduce how the edges in the set $\mathsf{WrongPairs}_{(P,(s \circ V),\prec_p,\prec_v)}$ are divided among the seven sets above. The numbers associated with the edges correspond with the set WP_s in which they reside. Edges in set WP^1_s, are edges between two nodes which are neither i nor j. Edges in WP^2_s are the incoming edges of i, and those incoming edges of j which result from the transitivity of $\prec_p$ on the incoming edges of i and the edge (i,j). Set WP^3_s consists of the outgoing edges of j, and those outgoing edges of i which result from $\prec_p$'s transitivity on the outgoing edges of j and the edge (i,j). WP^4_s contains the incoming edges of a node k, which is neither i nor j, and for which holds that $i \prec_p k \prec_p j$. Set WP^5_s is the singleton set with edge (i,j). WP^6_s comprises the edges (i,k), where k is not $\prec_p$-related to j (i.e. neither $k \prec_p j$ nor $j \prec_p k$ hold). Finally, WP^7_s is the set of the edges (k,j), where k is not $\prec_p$-related to i.

From 9.10 and 9.11, the validation of which are left as simple exercises to the reader, it follows that:

$$|\mathsf{WrongPairs}_{(P,(s \circ V),\prec_p,\prec_v)}| = |\mathsf{WP}^1_s| + |\mathsf{WP}^2_s| + |\mathsf{WP}^3_s| + |\mathsf{WP}^4_s| + \mathsf{WP}^5_s + |\mathsf{WP}^6_s| + |\mathsf{WP}^7_s| (9.12)$$

The proof of 9.9 now proceeds by comparing the cardinality of the different WP's in transition states s and t.

$|\mathsf{WP}^1_s| = |\mathsf{WP}^1_t|$, because the assumption $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$ indicates that the data values of processes other than i and j do not change.

$|\mathsf{WP}^2_s| = |\mathsf{WP}^2_t|$— In order to prove this equality, it suffices to show that there exists a bijection f from WP^2_s to WP^2_t. Below a function f is given which satisfies this constraint.

$f = \lambda(x,y).\textbf{if } y = i \textbf{ then } (x,j) \textbf{ else } (x,i)$.

The proof that f is a bijection is left as an exercise to the reader.

$|\mathsf{WP}^3_s| = |\mathsf{WP}^3_t|$— A similar proof as the one for $|\mathsf{WP}^2_s| = |\mathsf{WP}^2_t$— applies. A satisfactory bijection is:

$\lambda(x,y).\textbf{if } x = i \textbf{ then } (j,y) \textbf{ else } (i,y)$.

$|\mathsf{WP}^4_t| \leq |\mathsf{WP}^4_s|$— To prove this it is sufficient to verify that $\mathsf{WP}^4_t \subseteq \mathsf{WP}^4_s$, i.e. $\forall(x,y) :: (x,y) \in \mathsf{WP}^4_t \Rightarrow (x,y) \in \mathsf{WP}^4_s$. Suppose we have a pair $(x,y) \in \mathsf{WP}^4_t$, now we must show that $(x,y) \in \mathsf{WP}^4_s$. From the definition of WP^4_t we learn that there are two possibilities:

$\bullet_1$ $x = i$, $y \neq j$ and $\neg((t \circ V)i \prec_v (t \circ V)y)$.

Assuming these conditions, we must show that $(i,y) \in \mathsf{WP}^4_s$. Because we already have that $x = i$, $y \neq j$, $i \prec_p y$ and $y \prec_p j$, we only have to prove that $\neg((s\circ V)i \prec_v (s \circ V)y)$, which is equal to $(s \circ V)y \prec_v (s \circ V)i$ since $\prec_v$ is a total order. From the assumption that $\neg((t \circ V)i \prec_v (t \circ V)y)$ it can be deduced that:

1. $y \neq i$, because of the reflexivity of $\prec_v$ and the fact that $\neg((t\circ V)i \prec_v (t\circ V)y)$ holds.
2. $(t \circ V)y \prec_v (t \circ V)i$, for $\prec_v$ is a total order.

From the assumption $\mathsf{Swapped}_{(s,t,P,V,\prec_p,\prec_v)}$ and its presumed instantiation with (i,j), the following can be inferred:

3. $(t \circ V)i \prec_v (t \circ V)j$
4. $(t \circ V)j = (s \circ V)i$
5. $(s \circ V)y = (t \circ V)y$, because from 1 we can conclude that y is neither i nor j.

Now the transitivity and totality of $\prec_v$ establishes that $(s \circ V)y \prec_v (s \circ V)i$, since:

$$(s \circ V)y \overset{(5)}{=} (t \circ V)y) \overset{(2)}{\prec_v} (t \circ V)i \overset{(3)}{\prec_v} (t \circ V)j \overset{(4)}{=} (s \circ V)i$$

$\bullet_2$ $y = j$, $x \neq i$ and $\neg((t \circ V)x \prec_v (t \circ V)j)$.

In this case we must show that $(x,j) \in \mathsf{WP}^4_s$, which again comes down to showing that $(s \circ V)j \prec_v (s \circ V)x$ holds. Analogous to the previous proof this can be done by showing:

$$(s \circ V)j = (t \circ V)i \prec_v (t \circ V)j \prec_v (t \circ V)x = (s \circ V)x$$

.

$|\mathsf{WP}^5_t| < |\mathsf{WP}^5_s|$, because $|\mathsf{WP}^5_s|$ and $|\mathsf{WP}^5_t|$ equal 1 and 0 respectively.

$|\mathsf{WP}^6_t| \leq |\mathsf{WP}^6_s|$ **and** $|\mathsf{WP}^7_t| \leq |\mathsf{WP}^7_s|$, since $\bullet_1$ and $\bullet_2$ from the proof that $|\mathsf{WP}^4_t| \leq |\mathsf{WP}^4_s|$ show, that $\mathsf{WP}^6_t \subseteq \mathsf{WP}^6_s$ and $\mathsf{WP}^7_t \subseteq \mathsf{WP}^7_s$ respectively.

Since, the above seven items indicate that

$$\begin{aligned}&|\mathsf{WP}^1_t| + |\mathsf{WP}^2_t| + |\mathsf{WP}^3_t| + |\mathsf{WP}^4_t| + \mathsf{WP}^5_t + |\mathsf{WP}^6_t| + |\mathsf{WP}^7_t| < \\ &\quad |\mathsf{WP}^1_s| + |\mathsf{WP}^2_s| + |\mathsf{WP}^3_s| + |\mathsf{WP}^4_s| + \mathsf{WP}^5_s + |\mathsf{WP}^6_s| + |\mathsf{WP}^7_s|\end{aligned}$$

this completes the proof of 9.9, because, according to 9.12, this means:

$$|\mathsf{WrongPairs}_{(P,(t\circ V),\prec_p,\prec_v)}| < |\mathsf{WrongPairs}_{(P,(s\circ V),\prec_p,\prec_v)}|.$$

To complete the proof that the program satisfies specification 9.20, we still have to verify that the following:

1. $\forall p \in \mathsf{Conf}\colon\ p \in \mathsf{Pred}.(\mathbf{w}Sort)$
2. $(\ \vdash \Box\mathsf{Permutation}_{(P,V,V')})$

Verification of the first condition is left as an exercise to the reader, and the second condition can easily be proved by applying definition 6.17 and $\mathbf{S_{2c}}$.

9.6 Represent the program in the HOL embedding of UNITY

In order to represent the whole UNITY program from Figure 15 in HOL, we must start by defining the HOL-representation of processes, connections and networks of processes. Since processes can be uniquely identified by their labels, a process can be represented by its label. In HOL this can be established by:

```
let process = ":*pType";;
```

That is, the type variable `*pType` denotes the type of the processes's labels, and a process is modelled by something of this type. The ordering $\prec_p$ on the processes's labels and the ordering $\prec_v$ on the processes's data values now have types `process_Ord` and `value_Ord` respectively.

```
let process_Ord = ":^process -> ^process -> bool";;

let value_Ord = ":*val -> *val -> bool";;
```

Since connections are tuples of processes, they are modelled by the following type:

```
let connection = ": ^process # ^process";;
```

The function V that, given a process, returns the local variable of that process has the type `proc_vars` (which is short for process variables):

```
let proc_vars = ":^process -> *var";;
```

And the function that results from the composition of the function V with some state s has the following type:

```
let proc_vals = ":^process -> *val";;
```

Since a network of processes is a triple (P, C, V), where P is a set of processes, C is a set of connections and V a function of type `proc_vars`, the type of a network of processes is defined in HOL as:

```
let connections = ":(^connection)set";;
let processes = ":(^process)set";;

let network = ":^processes # ^connections # proc_vars";;
let Sort_PROG = new_definition
    (`Sort_PROG`,
     "Sort ((P, C, V):^network)
           (V':^proc_vals)
           (vOrd:^value_Ord)
           (pOrd:^process_Ord) =
     (CHF{Assign2 (V i, V j) (Min vOrd (V i) (V j),
                            Max vOrd (V i) (V j)) | (i IN P) /\ (j IN P) /\ (pOrd i j)
                                                  /\ (i,j) IN (Symmetric_Closure C)},
      (VALUES_EQ V V'),
      CHF{A (i:^process) | i IN P} ,
      CHF{A (i:^process) | i IN P})");;
```

Figure 17: The HOL definition of the program Sort.

Before the HOL representation of the UNITY program can be made, we have to resolve the overloading on the symbol = in the initial condition of the program in Figure 15. Following definition 9.11, we define the following in HOL:

```
let Values_EQ = new_definition
                 (`Values_EQ`,
                  "Values_EQ (V:^proc_vals) (V':^proc_vals)
                    = !(i:^process). (V i = V' i)");;
```

and the "lifted" version:

```
let VALUES_EQ = new_definition
                (`VALUES_EQ`,
                 "VALUES_EQ (V:^proc_vars) (V':^proc_vals)
                    = (\s:^State. Values_EQ  (s o V) V')");;
```

As an aside, throughout our HOL code we use the convention that for every state-dependent definition there are two HOL definitions, viz. one that defines the intended concept and a lifted version, written in small letters and capital letters respectively.

Now the construction of the HOL representation of the UNITY program is straightforward, and depicted in Figure 17.

9.7 Proof that the program is well-formed

To prove the well-formedness of the UNITY program, we must check the validity of the five conditions of which the predicate Unity consists.

$$\begin{aligned}\mathsf{Unity}.P \quad = \quad & (\forall a : a \in \mathbf{a}P : \Box_{\mathsf{En}} a) \;\wedge\; (\mathbf{w}P \subseteq \mathbf{r}P) \;\wedge \\ & (\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a) \;\wedge\; (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadleftarrow a)\end{aligned}$$

Whereas the first three conditions are easy to prove, the last condition, from now on called the read constraint, turns out to be [Pra95] very difficult to prove if a shallow embedding of the programming logic is not available. As promised in section 8 the UNITY embedding shall therefore be made deeper than the one presented in [Pra95]. Moreover, a tactic shall be presented that automatically proves the read constraint on an arbitrary syntactically correct (see page 20) UNITY program.

We start with defining the notion of invisibility for state-expressions, that is for some expression e and for some set V of variables we define a sufficient condition that states when the variables in V are invisible to e and hence do not influence e.

Definition 9.25 VARIABLES INVISIBLE-TO EXPRESSIONS *Exp_INVI_DEF*

$$V \not\rightarrow_e e$$
$$=$$
$$(\forall s, s' :: (s \restriction V^c = s' \restriction V^c) \Rightarrow \; e.s = e.s')$$

◀

Next, we formalise state-expressions in HOL. Below are some HOL definitions for conditional expressions (i.e. $\ldots \Longrightarrow \ldots \sim \ldots$), expressions consisting of only a variable, expressions for adding, and boolean expressions (i.e. state-predicates) that compare if their first argument is less-than their second argument.

```
let Expr = ":^State -> *val";;

let num_Expr = ":^State->num";;

let bool_Expr = ":^State->bool";;

let VAR_EXPR_DEF = new_definition
                  (`VAR_EXPR_DEF`,
                   "VAR_EXPR (v:*var) = (\(s:^State). (s v))");;

let COND_EXPR_DEF = new_definition
                   (`COND_EXPR_DEF`,
                    "COND_EXPR (P:^bool_Expr) (E1:^Expr) (E2:^Expr)
                                   = (\s. (P s) => (E1 s) | (E2 s))");;

let ADD_EXPRS_DEF = new_infix_definition
                  (`ADD_EXPRS_DEF`,
                   "ADD_EXPRS (E1:^num_Expr) (E2:^num_Expr)
                         = (\(s:^State). (E1 s) + (E2 s))");;

let LT_DEF = new_infix_definition
              (`LT_DEF`,
                "LT (E1:^num_Expr) (E2:^num_Expr) = (\(s:^State). (E1 s) < (E2 s))");;
```

Now we must construct and prove theorems that state when these expressions are invisible to a set of variables, see Figure 18. Since these theorems will be used to prove the read constraint (i.e. $(\forall a : a \in \mathbf{a}Sort : (\mathbf{r}Sort)^c \not\rightarrow_e a))$, they state when expressions are invisible to the *complement* of a set of variables.

Finally we can create theorems that state when actions are invisible to the complement of a set of variables, see Figure 19. Theorem 9.30 expounds when a simultaneous assignment action, the definition of which was given in 6.7 on page 18 and the HOL representation of which was presented in section 7.2, is invisible to the complement of some set. Other single

For some set R of variables:

Theorem 9.26 *VAR_EXPR_SAT_RC*
Let v be a variable.

$$\frac{(R\,v)}{(R^{\mathsf{c}} \not\twoheadrightarrow_e (\lambda s.(s\,v)))}$$

Theorem 9.27 *COND_EXPR_SAT_RC*
Let P be some state-predicate (i.e. boolean expression), and let E_1 and E_2 be two state-expressions:

$$\frac{(R^{\mathsf{c}} \not\twoheadrightarrow_e P) \wedge (R^{\mathsf{c}} \not\twoheadrightarrow_e E_1) \wedge (R^{\mathsf{c}} \not\twoheadrightarrow_e E_2)}{(R^{\mathsf{c}} \not\twoheadrightarrow_e (\lambda s.P.s \implies E_1.s \sim E_2.s))}$$

Theorem 9.28 *ADD_EXPRS_SAT_RC*
let E_1 and E_2 be two state-expressions:

$$\frac{(R^{\mathsf{c}} \not\twoheadrightarrow_e E_1) \wedge (R^{\mathsf{c}} \not\twoheadrightarrow_e E_2)}{(R^{\mathsf{c}} \not\twoheadrightarrow_e (\lambda s.E_1.s + E_2.s))}$$

Theorem 9.29 *ALL_ORD_GUARD_SAT_RC*
let E_1 and E_2 be two state-expressions, and let $\prec$ be some ordering on elements with values from Val:

$$\frac{(R^{\mathsf{c}} \not\twoheadrightarrow_e E_1) \wedge (R^{\mathsf{c}} \not\twoheadrightarrow_e E_2)}{(R^{\mathsf{c}} \not\twoheadrightarrow_e (\lambda s.E_1.s \prec E_2.s))}$$

Figure 18: Invisibility of state-expressions.

actions have similar theorems, we only present this one here since it is used in our sorting program from figure 17. Theorem 9.31 expresses the conditions that have to be satisfied in order to show invisibility of a guarded action.

With all se theorems we can easily prove that the read constraint is satisfied for a syntactically correct UNITY program. For example, for a UNITY program of which the assignments can only assign to two variables at the same time, the following tactic immediately proofs the read constraint:

```
REPEAT
  (PROVE_IS_ELT_TAC
   ORELSE MATCH_MP_TAC Assign2_SAT_RC
   ORELSE MATCH_MP_TAC Guard_Action_SAT_RC
   ORELSE MATCH_MP_TAC COND_EXPR_SAT_RC
   ORELSE MATCH_MP_TAC ALL_ORD_GUARD_SAT_RC
   ORELSE MATCH_MP_TAC ADD_EXPRS_SAT_RC
   ORELSE MATCH_MP_TAC VAR_EXPR_SAT_RC
   ORELSE REPEAT CONJ_TAC
   )
```

For some set R of variables:

Theorem 9.30 *Assign2_SAT_RC*
Let $x, y \in \mathsf{Var}$, let E_1 and E_2 be two state-expressions, and let R be a set of variables (i.e. $R \subseteq \mathsf{Var}$).

$$\frac{(R\,x) \wedge (R\,y) \wedge (R^{\mathsf{c}} \not\rightarrow_e E_1) \wedge (R^{\mathsf{c}} \not\rightarrow_e E_2)}{(R^{\mathsf{c}} \not\rightarrow \mathsf{assign}_2.(x,y).(E_1, E_2))}$$

Theorem 9.31 *Guard_Action_SAT_RC*
Let g be some state-predicate (i.e. boolean-expression), let a and b be two actions, and let R be a set of variables (i.e. $R \subseteq \mathsf{Var}$).

$$\frac{(R^{\mathsf{c}} \not\rightarrow_e g) \wedge (R^{\mathsf{c}} \not\rightarrow a) \wedge (R^{\mathsf{c}} \not\rightarrow b)}{(R^{\mathsf{c}} \not\rightarrow \ \mathsf{if}\ g\ \mathsf{then}\ a\ \mathsf{else}\ b)}$$

Figure 19: Invisibility of actions

where `PROVE_IS_ELT_TAC` is a tactic that proves that a variable is an element of the declared read variables. Applied to a read constraint, this tactic basically works as a loop (see Figure 9.7). During the first iteration, it tries to match one of the the conclusions of theorems 9.26 up to and including 9.31 with the proof goal, thereby generating a new subgoal. Looking at the hypothesis of theorems 9.26 up to and including 9.31 we see that this generated subgoal can take three forms:

1. $(R\,v)$, i.e. it must be proved that a variable is a declared read variable.
2. another invisibility predicate, i.e. $(R^{\mathsf{c}} \not\rightarrow_e e)$ or $(R^{\mathsf{c}} \not\rightarrow a)$
3. a conjunction of forms 1 and 2.

During the subsequent iterations the tactic is repeatedly applied to the generated subgoals. Subgoals of the form (1) are immediately proved by the tactic `PROVE_IS_ELT_TAC`. Subgoals of form (2) or (3), cause the generation of new subgoals. From a subgoal g of form (2), a new subgoal of form (1), (2) or (3) is generated by matching g to one of the the conclusions of theorems 9.26 up to and including 9.31. From a subgoal g of form (3), new subgoals of forms (1) or (2) are generated by `REPEAT CONJ_TAC`, which repeatedly splits the conjunctive goal g into its different conjuncts. Since, ultimately all generated subgoals shall have form (1), the tactic eventually proves every conjecture of the form $(R^{\mathsf{c}} \not\rightarrow a)$.

Although we now have a tactic which automatically proves the read constraint on the UNITY program of Figure 17, what we really want is a tactic that proves the read constraint for every syntactically correct UNITY program, i.e. UNITY programs whose assignment actions can assign values to an arbitrary number of variables at the same time.

The way in which we currently represent assignment actions in HOL is by a separate HOL definition for every number n, $n > 0$, where n denotes the number of variables to which some value is assigned by the assignment statement. Consequently, in order to be

```
goals = { read-constraint }

while there still exists a $g \in goals$
do

  if $g$ has the form $(R^c \not\twoheadrightarrow_e e)$ or $(R^c \not\twoheadrightarrow a)$
  then match $g$ to one of the conclusions of theorems 9.26 up to and including
       9.31, and add the hypothesis of the matching theorem to the set goals.

  if $g$ has the form $R\,v$
  then prove goal $g$ by PROVE_IS_ELT_TAC

  if $g$ is a conjunction
  then split $g$ into its different conjuncts by REPEAT CONJ_TAC
       and add these to the set goals

  goals = goals $-g$

od
```

Figure 20: Proving a read constraint.

able to automatically verify the read constraint of a UNITY program which also contains assignment actions that assign to three variables at the same time, we must:

- add a new HOL-definition for "triple-assignment actions":

$$\begin{array}{c}\mathsf{assign}_3.(x_1,x_2,x_3).(E_1,E_2,E_3)\\ =\\ (\lambda s,t.\ t.x_1 = E_1.s)\ \sqcap\ (\lambda s,t.\ t.x_2 = E_2.s)\ \sqcap\ (\lambda s,t.\ t.x_3 = E_3.s)\ \sqcap\ (\mathsf{skip}\lceil\{x_1,x_2,x_3\}^c)\end{array}$$

- prove the following theorem:

Theorem 9.32 *Assign3_SAT_RC*
Let R be a set of variables, let x_1, x_2 and x_3 be three variables, and let E_1, E_2 and E_3 be three state-expressions.

$$\frac{(R^c \not\twoheadrightarrow_e E_1) \wedge (R^c \not\twoheadrightarrow_e E_2) \wedge (R^c \not\twoheadrightarrow_e E_3) \wedge (R\,x_1) \wedge (R\,x_2) \wedge (R\,x_3)}{(R^c \not\twoheadrightarrow \mathsf{assign}_3.(x_1,x_2,x_3).(E_1,E_2,E_3))}$$

- change our tactic by adding the following line to the inner tactic:

```
ORELSE MATCH_MP_TAC Assign3_SAT_RC
```

Evidently this is not very efficient. Moreover, it will not be possible to construct a tactic which shall automatically prove the read constraint for an arbitrary UNITY program, since a limit on the number of variables to which can be assigned at the same time must always be made. Consequently, we must change our representation of assignment actions, and come up with one definition for every possible assignment statement. Obviously, this can be achieved by using lists of variables and lists of expressions, since lists can contain arbitrary numbers of elements.

In the new definition of assignment actions, which we are about to give, some well-known functions that stem from Functional Programming (a good introduction to which can be found in [BW88]) are used. The definitions of these functions are given below by pattern matching with `[]` and `:`. Here `[]` denotes the empty list, and `:` is the cons operator on lists which inserts a value as a new first element of a list, e.g. `1:[2,3] = [1,2,3]`.

```
map f []     = []
map f (x:xs) = (f x) : map f xs
```

For instance, `map (+2) [1,2,3,4] = [3,4,5,6]`.

```
zip [] ys         = []
zip xs []         = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

For instance, `zip [1,2,3] [4,5,6] = [(1,4), (2,5), (3,6)]`.

```
foldr op e []     = e
foldr op e (x:xs) = x op (foldr op e xs)
```

For instance, `foldr + 0 [1,2,3,4] = 1 + 2 + 3 + 4 + 0`. Now we can give the definition of an assignment action;

$$\begin{aligned}\mathsf{assign}.lv.le \;=\;& (\lambda s\, t.(\#lv = \#le) \wedge (\#le > 0)) && (9.13)\\ & \sqcap \\ & \mathtt{foldr}(\sqcap)(\lambda s\, t.\, true)(\mathtt{map}\;\mathtt{update}\,(\mathtt{zip}\; lv\; le)) && (9.14)\\ & \sqcap \\ & (\mathsf{skip} \restriction (\lambda x.\neg(x \in_l lv))) && (9.15)\end{aligned}$$

where $x \in_l xs$ denotes that x is an element in the list xs; `#` is an operator which given a list, returns the number of elements in that list; and `update` is the following function:

$$\mathtt{update}\, x\; e = (\lambda s\, t.\, t.x = e.s)$$

9.13 states that the number of variables at the left hand side of an assignment should be equal to the number of expressions at the right hand side. Moreover, it expresses that an assignment action should assign a value to at least one variable.

9.14 describes the state-transition the assignment action should make. In other words, it defines the values of the variables in the new state.

9.15 argues that only that values of the variables that occur at the left hand side of the assignment change, i.e. variables not present in the list lv are left intact.

Furthermore, we need the following theorem, which states the condition an assignment action must satisfy in order for it to be invisible to the complement of a set of variables.

Theorem 9.33 *Assign_SAT_RC*
Let R be a set of variables, let lv be a list of variables, and let le be a list of state-expressions.

$$\frac{(\forall x \in_l lv\ :\ (R\,x)) \wedge (\forall E \in_l le\ :\ R^{\mathsf{c}} \not\rightarrow_e E)}{(R^{\mathsf{c}} \not\rightarrow \mathsf{assign}.lv.le}\,)$$
◀

As a result, the tactic which automatically proves the read constraint for an arbitrary UNITY program is:

```
REPEAT
  (PROVE_IS_ELT_TAC
   ORELSE (MATCH_MP_TAC Assign_SAT_RC
           THEN CONVERT_INTO_CORRECT_FORM_TAC)
   ORELSE MATCH_MP_TAC Guard_Action_SAT_RC
   ORELSE MATCH_MP_TAC COND_EXPR_SAT_RC
   ORELSE MATCH_MP_TAC ALL_ORD_GUARD_SAT_RC
   ORELSE MATCH_MP_TAC ADD_EXPRS_SAT_RC
   ORELSE MATCH_MP_TAC VAR_EXPR_SAT_RC
   ORELSE REPEAT CONJ_TAC
   )
```

Note that we have just substituted the tactic `MATCH_MP_TAC Assign2_SAT_RC` by the tactic `MATCH_MP_TAC Assign_SAT_RC THEN CONVERT_INTO_CORRECT_FORM_TAC`.

The tactic `CONVERT_INTO_CORRECT_FORM_TAC` is a tactic which converts the hypothesis of Theorem 9.33 into subgoals of the forms (1), (2) or (3) (see page 67).

9.8 Prove that the program satisfies the specification

In order to prove that the program satisfies its specification, we must construct a HOL-proof which verifies the following theorem:

Theorem 9.34 PROGRAM *Sort* SATISFIES THE SPECIFICATION *SORTING_PROG_SPECIFICATION*

$$\frac{\begin{array}{c}\mathsf{Network}(P,C,V) \wedge \mathsf{Total}(\prec_v, V) \wedge (\prec_p \neq \emptyset) \\ \mathsf{AntiSymmetric}(\prec_p, P) \wedge \mathsf{Transitive}(\prec_p, P) \wedge \mathsf{SufficientConnections}_{(C,\prec_p)}\end{array}}{\mathsf{Permutation}_{(P,V,V')}\ {}_{Sort}\vdash (V = V') \rightsquigarrow \mathsf{Sorted}_{(P,V,\prec_p,\prec_v)}}$$
◀

Until now, we have not yet seen a formal definition of the predicate $\mathsf{Network}(P,C,V)$. Since we need the HOL definition of this predicate in order to represent the above theorem in HOL, this definition is presented in Figure 21. Because a network of processes is represented as an undirected graph, the HOL definitions for graphs are also included in Figure 21.

```
let direction_Axiom =
    define_type `direction_Axiom` `direction = DIRECTED | UNDIRECTED`;;

let edge = ":*vType#*vType";;

let vertex = ":*vType";;

let edges = ":(^edge)set";;

let vertices = ":(^vertex)set";;

let info = ":(^vertex -> *infoType)";;

let graph = ":^vertices # ^edges # ^info # direction";;

let Edges = new_definition(`Edges`,
    "Edges ((V, E, f, D):^graph) = E");;

let Vertices = new_definition(`Vertices`,
    "Vertices ((V, E, f, D):^graph) = V");;

let Info = new_definition(`Info`,
    "Info ((V, E, f, D):^graph) = f");;

let Direction = new_definition(`Direction`,
    "Direction ((V, E, f, D):^graph) = D");;

let POSSIBLE_EDGES_DEF = new_definition (`POSSIBLE_EDGES_DEF`,
   "POSSIBLE_EDGES (V:(*vType)set) (D:direction) =
    (D = DIRECTED) => {(u,v) | (u IN V) /\ (v IN V)}
                    | {(u,v) | (u IN V) /\ (v IN V) /\ ~(u=v)}");;

let Graph_DEF = new_definition (`Graph_DEF`,
    "Graph (G:^graph)
         = (FINITE (Vertices G))
           /\ (~((Vertices G) = {}))
           /\ ((Edges G) SUBSET (POSSIBLE_EDGES (Vertices G) (Direction G)))");;

let Conns = new_definition (`Conns`,
   "Conns ((P, C, V):^network) = C");;

let Procs = new_definition (`Procs`,
   "Procs ((P, C, V):^network) = P");;

let Vals = new_definition (`Vals`,
   "Vals ((P, C, V):^network) = V");;

let Network_DEF = new_definition (`Network_DEF`,
   "Network (NW:^network)
      = Graph (Procs NW, Conns NW, Vals NW, UNDIRECTED) /\
        (!(i:^proces) (j:^proces). (~(i=j)) ==> (~(A i = A j)))");;
```

◀

Figure 21: HOL definitions for graphs and networks.

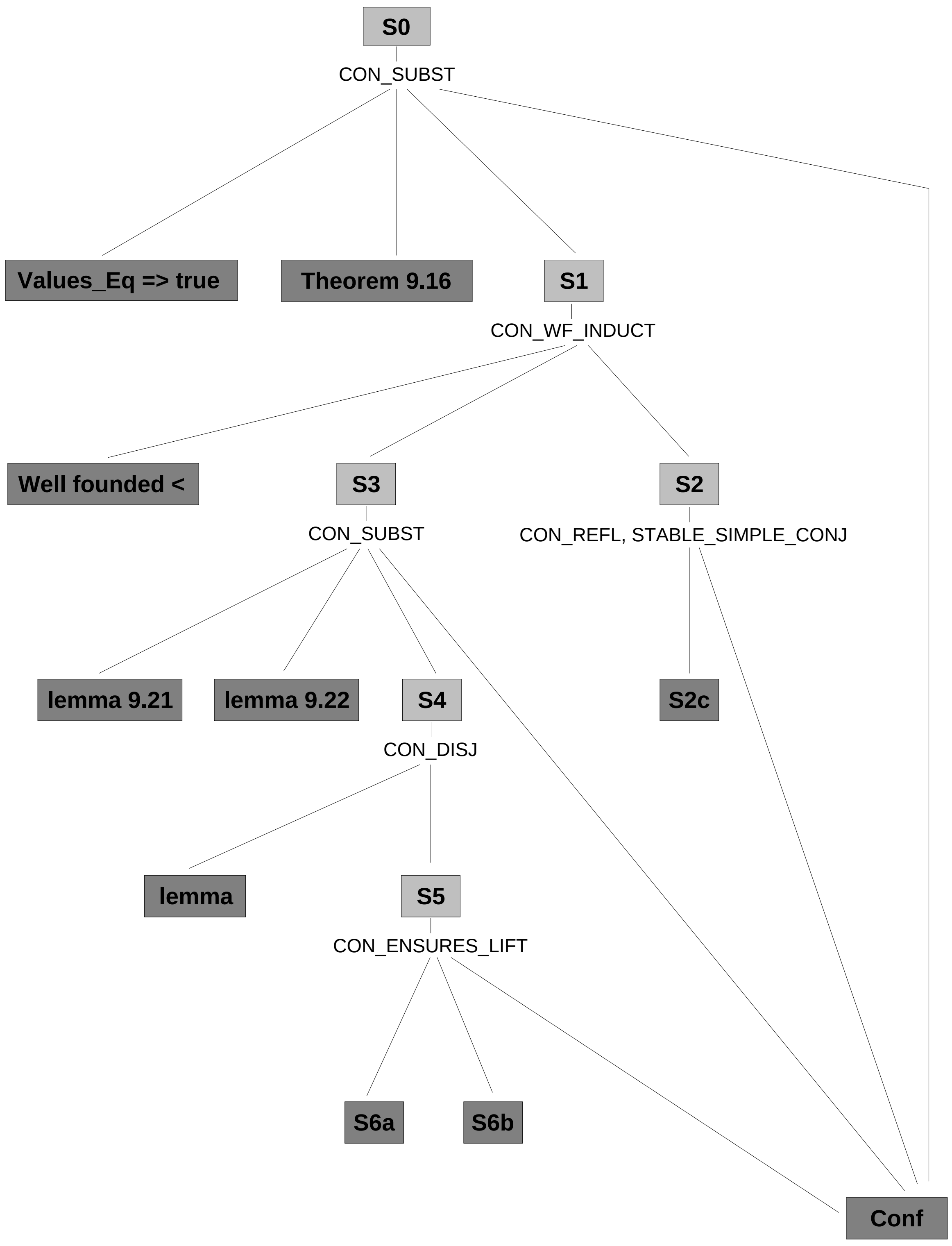

Figure 22: The proof tree which must be constructed to prove that the program satisfies the specification.

The HOL term which represents specification 9.34 is:

```
"!(P:^processes) (C:^connections) (V:^proc_vars)
 (pOrd:^proces_Ord) (vOrd:^value_Ord).

  ((Network (P,C,A))
  /\ (Sufficient_Connections_In_Network_For_Sorting pOrd (P,C,A))
   /\ (SORT_ORDER vOrd P A) /\ (ANTI_SYMM pOrd P) /\ (TRANSITIVE pOrd P)
   /\ (NOT_EMPTY pOrd)

   ==>

  !(V':^proc_vals).
   (CON ((Sort (P, C, V) V' vOrd pOrd):
            (((*var -> *val) -> ((*var -> *val) -> bool)) -> bool)
            # (((*var -> *val) -> bool)
            # ((*var -> bool)
            # (*var -> bool))))

    (PERM P A A')

    (VALUES_EQ A A')

    (SORTED P A vOrd pOrd))"
```

where the HOL definition of `SORT_ORDER vOrd P A` is:

```
let SORT_ORDER_DEF = new_definition
                  (`SORT_ORDER_DEF`,
                   "SORT_ORDER (vOrd:^value_Ord) (P:^processes) (A:^proc_vars) =
                   !(s:^State). TOTAL_ORDER vOrd {(s o A) i | i IN P}");;
```

Note that this additional definition is necessary because overloading like $(V\ i) \prec_v (V\ j)$ on the symbol $\prec_v$ is not possible in HOL. Consequently, we must define $\prec_v$ to be a total order on $(s \circ V)$ for every state s explicitly.

In order to prove the theorem, a proof tree must be constructed using the refinement and decomposition method presented in section 9.4. Figure 22 shows the resulting proof tree. The UNITY inference rules which were applied in section 9.4, are denoted by their HOL equivalents. At the root of this proof tree we find the specification $\mathbf{S_0}$, and at the leaves are the proof obligations among which $\mathbf{S_{2c}}$, $\mathbf{S_{6a}}$, $\mathbf{S_{6b}}$ and Conf. To prove that the program meets the specification, we must build a closed proof tree, which boils down to proving all the conjectures at the leaves, and construct the tree by backward or forward proof. Since HOL provides better support for backward proof than it does for forward proof, and as well as that has the facility to interactively construct a backward proof we shall choose to construct the tree by backward proof, that is from top to bottom.

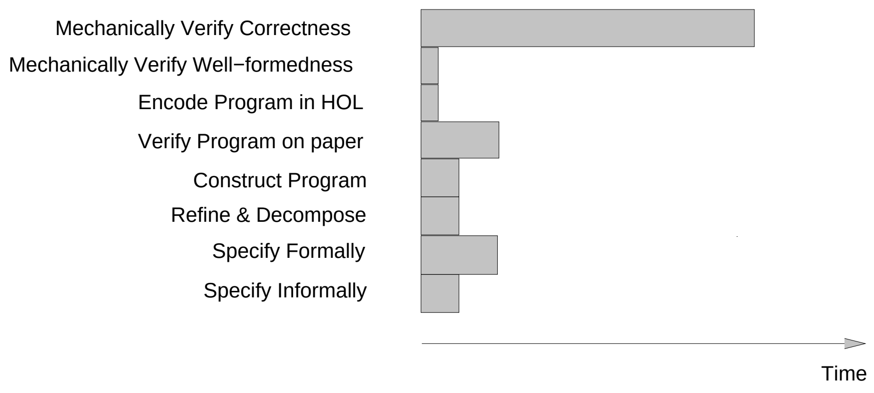

Figure 23: A rough, proportionate estimate of the time spend on the individual steps from section 8.

10 Reflections

Whoever destroys a single life is as guilty as though he has destroyed the whole world; and whoever rescues a single life earns as much merit as though he had rescued the entire world.
–Sanhedrin 37:a
The Talmud

Sketched in broad outlines, our experiences with formal methods and theorem provers used during the development process of (distributed) programs are twofold:

1. They are hard to sell, especially to industry.
2. Using them takes time.

In the light of experience 1, we would like to compare constructing safety-critical software with driving a car. Within this context, we collate driving a car under the influence of alcohol to constructing safety critical software *without* the use of formal methods and mechanical verification.

Even without alcohol car crashes can and do happen, there is no getting away from that. All human activity involves risks, and the only way in which "risk-free traffic" could be achieved is by prohibiting every form of transportation altogether. Needless to say, a very unlikely thing to happen. Nevertheless, it is nowadays widely accepted that driving under the influence of alcohol significantly augments the risk of car accidents, and as a result intoxicated driving is prohibited by law. Unfortunately, there are still people that do not restrain themselves from intoxicated driving. By pointing out the dangers to them, frequently heard defences are: "Yes, but *I* can drive very well", or, "*I* am always very careful when I drink and drive.". Everyone shall agree that these are not very solid pleas.

Similar arguments can be made, however, for formal methods and mechanical verification and their use in the development of safety-critical software. Although we are convinced that formal methods and mechanical verification of safety critical software are inescapably necessary, we do not pretend as though using them shall solve all problems and make software inherently safe. Moreover a safe alternative may not always be available (the safest air-plane is the one that never leaves the ground). Even when formal methods and mechanical verification are used software can be incorrect since the real world is not a formal system. A proof, therefore, does not show that in the real world things shall happen as one expects. All engineering disciplines, however, are concerned with making models of the real world, there is no getting away from that. Needless to say, nothing can achieve perfection. Nevertheless, many believe that formal methods currently offers the only intellectually defensible method for increasing the reliability of safety-critical software. Unfortunately, most of the safety-critical software is constructed without using any formal method not to mention mechanical verification. By pointing out the dangers, frequently heard defences are "Well, *we* are always very accurate in distinguishing all cases", or, "Yes, but *we* always test very well.". Obviously, considering the increasing complexity of (safety-critical) software, and an often quoted remark that "Program testing can be used to show the presence of bugs, but never their absence."[DDH72, page 6], these defences are by no means adequate for not being formal.

In the light of experience 2, formally developing and mechanically verifying the sorting program took approximately 6 man months, which included

- getting acquainted with HOL, UNITY and Prasetya's embedding of the latter in HOL.
- working through all the steps described in section 8. (see Figure 23)

This is a great amount of time, considering the complexity of the problem solved. Looking back, it can be said, with a vast degree of certainty, that half of this time was spend on the machine-checked proofs.

Two important questions arise:

"Are these large amounts of time justifiable ?".

"Are these large amounts of time acceptable in industrial applications?"

The answer to the first question is definitely yes. If only one single human life can be spared as a result of applying formal methods and mechanical verification to the development of safety critical software, then any amount of time that is spend on mechanical formal verification is justifiable. The work of Muffy Thomas [Tho94a, Tho94b] uncovered several errors in the software which was used in the Therac-25, by using formal specification and verification techniques.

Obviously, the answer to the second question is no. Consequently, one important goal of future research in the area of formal methods and mechanical verification must be the development of tools and techniques that reduce the time which is necessary to formally and mechanically develop and verify safety critical software. In 1994 a so-called HOL2000 initiative was created, with the goal to try to put together a design for the next generation of HOL-like theorem proving environments. The latter should not be a redesign from zero, but an evolution from the current state using the accumulated wisdom among HOL users about HOL's strengths and weaknesses. Some issues that have been suggested are:

Better user interface. The obvious motivation for better user interfaces to HOL are simple: HOL is an excellent theorem proving system, but is has a crude, command-line interface. The following list of possible benefits, which is drawn from [Sym95], indicates how HOL users can benefit from a good graphical user interface (GUI).

- Much better ways of visualising theory development. The current interface to HOL is weak, particularly in the area of visualisation.
- HOL will get more accessible.
- HOL will be easier to learn with a good GUI. The present text interface provides little help in reducing the steepness of the learning curve.
- Users expect high quality data presentation in the tools they use. The command-line user interface to HOL does not currently provide this.

Currently, the following user-interfaces are available:

- a HOL and the Emacs editor. Here the Emacs editor is used as an interface to HOL.
- An X Windows based graphical Interface to HOL, called xhol [SB94]. Interaction with HOL is accomplished through an emacs-style input window, a utility toolbar, and a collection of windows that display information about the state of the proof. Featured is a graphical display of the entire proof tree. This display does not only presents a road map of what has and has not been proven, it also provides the user with clues about what techniques or tactics may be useful in proving the remaining unsolved subgoals.
- An X-based User Interface to HOL called CHOL [Thé93]. CHOL offers a "better user-interface for the HOL theorem prover". It has been developed following a general approach for building user-interfaces for theorem provers [TBK92]. Its characteristics are the following: Graphic interface: Multi fonts and colours are used to display formulas and commands; Structured editing: The proof of a proposition is presented as the structured editing of the tactic that proves the proposition; Rewriting tool: A tool has been developed that filters the theorems of the database with respect to the term to rewrite; Separation between the interface and the prover: The X interface and HOL run as two separate processes.
- a Tcl/Tk User Interface to HOL. TkHOL [Sym95] is an interface for HOL90 implemented using the Tk/Tcl toolkit. It is an extensible set of tools for viewing and browsing theories, searching the theory hierarchy for particular theorems, performing backward proofs and making (recursive type) definitions. The command line HOL interface is still available.

Improved error messages, which give more clues as to where the error can be sought, and might even give hints as to what to correct.

More libraries and accompanying tools. The HOL system's library facility provides a repository for reusable tools and theories. Each library is a self-contained piece of work with its own documentation, written according to a prescribed standard. The standard requires that there be a reference manual for the library which is compatible with the main HOL Reference Manual. A selection of libraries of the HOL system can be found on http://lal.cs.byu.edu/lal/holdoc/library.html. The UNITY embedding described in this report is an example of a library for HOL, an other UNITY library is also available [And92]. An other example is the `arith` library by Boulton [Bou94], the main tool of which is a partial decision procedure, `ARITH_CONV`, for Presburger natural number arithmetic [Coo72]. This tool enables users of the HOL system to automatically prove arithmetic lemmas in a practical amount of time.

The availability of all sorts of libraries are valuable in theorem provers. If, for example, the UNITY library was not available we would not have been able to verify our convergent distributed sorting program in six months. Because, if we insisted on using UNITY as a specification language, we ourselves ought to have created the embedding of UNITY within HOL, which would have cost us much more than six months. So, the availability of a HOL embedding of some specification language can significantly reduce the time which is spend on the mechanical verification of software specified within that particular language. Consequently, we shall need libraries for various specification languages, so that the software developer can choose among many specification languages and pick that one which relates most to the characteristics of the specific product being developed. This will also make HOL more accessible since the unavailability of one's favourite and most suitable specification language's embedding in HOL is no longer a barrier for using HOL.

Decision procedures are an important tool in theorem provers. They allow much low-level reasoning to be performed automatically. Lemmas and theorems that appear trivial may take many minutes or even days to prove by hand, especially for inexperienced users. Decision procedures can relieve users of some of this burden. `ARITH_CONV`, for example, was very useful to us, and saved us lots of time. Unfortunately, the HOL system suffers from a relative lack of decision procedures, in addition to the Presburger procedure, the `taut` library, written by [], provides a decision procedure for propositional tautologies, the library `faust`, written by Schneider, Kropf, and Kumar [SKR91], provides a decision procedure to check the validity of many formulae from first order predicate logic. A limitation of decision procedures is that formulae to be proved rarely conform to one theory. Usually, these formulae involve symbols from other theories for which there may or may not be a decision procedure available. Sometimes, the replacement of such "non-conforming" symbols with new variables does not affect the truth of the formula, and consequently proving the generalised formula followed by re-instantiation is successful. Often, however, the decision procedure cannot be used. For this reason there is much interest in decision procedures for combinations of theories [Bou95].

More efficiency HOL is a fully-expansive theorem prover [Bou93], which means that proofs generated in the system are composed of applications of the primitive inference rules of the underlying logic. The advantages of a fully-expansive theorem prover are twofold. Firstly, the soundness of the system depends only on the implementations of the primitive inference rules. Secondly, users are free to write their own proof procedures without the risk of making the system unsound. The disadvantage, however, is that the insistence on resolving proofs into simple primitive inferences can make HOL slow, and thus performance is compromised. The work of Richard Boulton [Bou93] focuses on the improvement of fully-expansive theorem provers by eliminating duplicated and unnecessary primitive inferences.

Combining theorem proving with model checking. Model checking is a very successful technique for automatically verifying finite-state programs. A model-checking algorithm exhaustively traverses all possible executions of a program, extensively enough to be able to answer a given question about the program's behaviour. The advantage is that we are relieved from the pain of constructing proofs, because the whole process is automated. A disadvantage, however, is that this technique only works for programs with a finite state-space, and, unfortunately, state spaces of most safety-critical programs are infinite. Since mechanical verification techniques can cope with infinite state spaces, an interesting topic is to extend or combine interactive theorem provers with various automatic tools such as model checkers. [KKS91, Bou92, Bus94].

Time can not be reduced for ever, there must always be some amount of time which shall be necessary to understand the problem to be solved completely.

A HOL definitions and pre-proved theorems

This appendix provides lists of definitions and verified theorems as they are in HOL. Not all definitions and theorems that we produced will be listed —it would take too much space. Only those we consider relevant or interesting for the reader will be included. As warned previously, there will be a discrepancy between the HOL definitions and theorems and the 'hand' versions as presented in earlier chapters.

A.1 Relations, orderings and permutations

Some standard definitions on relations, orderings and permutations. Note that in HOL relations are modelled by predicates.

```
p2s
  /* Converts a predicate P over two sets S1
     and S2 (i.e. P has type S1->S2->bool) into
     the following set:
    {(x,y)|(x in S1) /\ (y in S1) /\ (P x y)} */

  |- !Ord. o2s Ord = SPEC(UNCURRY Ord)

CARTHES_PROD_DEF
  |- !Set1 Set2.
      CARTHES_PROD Set1 Set2
      = {(x,y) | x IN Set1 /\ y IN Set2}

RELATION_DEF
  |- !R Set1 Set2.
      RELATION R Set1 Set2
       = (p2s R) SUBSET (CARTHES_PROD Set1 Set2)

ORDERING_DEF
  |- !R Set. ORDERING R Set = RELATION R Set Set

REFL_ORD_DEF
  |- !Ord Set.
      REFL_ORD Ord Set = (!x. x IN Set ==> Ord x x)

ANTI_REFL_ORD_DEF
  |- !Ord Set.
     ANTI_REFL_ORD Ord Set
         = (!x. x IN Set ==> ~Ord x x)

ANTI_SYMM_ORD_DEF
  |- !Ord Set.
      ANTI_SYMM_ORD Ord Set =
      (!x y. y IN Set /\ x IN Set
          ==> ~(x = y) ==> ~(Ord x y /\ Ord y x))

SYM_ORD_DEF
  |- !Ord Set.
      SYM_ORD Ord Set =
      (!x y. y IN Set /\ x IN Set
              ==> Ord x y ==> Ord y x)

TRANSITIVE_ORD_DEF
  |- !Ord Set.
      TRANSITIVE_ORD Ord Set =
      (!x y z.
        y IN Set /\ x IN Set /\ z IN Set ==>
        Ord x y /\ Ord y z ==>
              Ord x z)

PARTIAL_ORDER_DEF
  |- !Ord Set.
      PARTIAL_ORDER Ord Set =
      ORDERING Ord Set /\
      REFL_ORD Ord Set /\
      ANTI_SYMM_ORD Ord Set /\
      TRANSITIVE_ORD Ord Set

TOTAL_ORDER_DEF
  |- !Ord Set.
      TOTAL_ORDER Ord Set =
      PARTIAL_ORDER Ord Set /\
      (!x y. y IN Set /\ x IN Set
            ==> Ord x y \/ Ord y x)

SORT_ORDER_DEF
  |- !vOrd P A.
    SORT_ORDER vOrd P A
      = (!s. TOTAL_ORDER vOrd{(s o A)i | i IN P})

INDEX_ORDER_DEF
  |- !Ord Set.
    INDEX_ORDER Ord Set =
    RELATION Ord Set /\ ANTI_SYMM_REL Ord Set /\
    TRANSITIVE_REL Ord Set

Symmetric_Closure
   /*Here Ord is represented as a set*/

  |- !Ord. Symmetric_Closure Ord
        = Ord |_| {(u,v) | Ord v u}

Transitiv_Refl_Closure
   /*Here Ord is represented as a set*/

  |- !Ord Set.
      Transitive_Refl_Closure Ord Set =
      SPEC
      (\p.
        ?n f.
         (!i. i <= n ==> (f i) IN Set) /\
         (f 0 = FST p) /\ (f n = SND p) /\
         (!i. i < n ==> (f i, f(SUC i)) IN Ord))

Values_EQ
 |- !A A'. Values_EQ A A' = (!i. A i = A' i)
```

```
Perm_DEF
  |- !P A A'.
      Perm P A A' =
```

```
(?f.
  (!i. i IN P ==> (f i) IN P) /\
  BIJECTION f /\
  (!i :: \i. i IN P. A i = A'(f i)))
```

A.2 Graphs and networks

```
Edges  |- !V E f D. Edges(V,E,f,D) = E

Vertices  |- !V E f D. Vertices(V,E,f,D) = V

Info  |- !V E f D. Info(V,E,f,D) = f

Direction  |- !V E f D. Direction(V,E,f,D) = D

POSSIBLE_EDGES_DEF
  |- !V D.
      POSSIBLE_EDGES V D =
      ((D = DIRECTED) =>
       {(u,v) | u IN V /\ v IN V} |
       {(u,v) | u IN V /\ v IN V /\ ~(u = v)})

Graph_DEF
  |- !G.
      Graph G =
      FINITE(Vertices G) /\
      ~(Vertices G = {}) /\
      (Edges G) SUBSET
         (POSSIBLE_EDGES(Vertices G)(Direction G))

is_Edge_DEF
  |- !u v E D.
      is_Edge(u,v)E D =
```

```
      ((D = DIRECTED) => (u,v) IN E
                      | ((u,v) IN E \/ (v,u) IN E))

is_Vertex_DEF  |- !v V. is_Vertex v V = v IN V

Conns  |- !P C V. Conns(P,C,V) = C

Procs  |- !P C V. Procs(P,C,V) = P

Vals  |- !P C V. Vals(P,C,V) = V

POSSIBLE_CONNECTION_DEF
  |- !P.
      POSSIBLE_CONNECTIONS P
       = {(i,j) | i IN P /\ j IN P /\ ~(i = j)}

Network_DEF
  |- !NW. Network NW
      = Graph(Procs NW,Conns NW,Vals NW,UNDIRECTED)

Network_IMP_FINITE_PROCESSES
  |- !NW. Network NW ==> FINITE(Procs NW)

Network_IMP_PROCESSES_NOT_EMPTY
  |- !NW. Network NW ==> ~(Procs NW = {})
```

A.3 Predicate Operators

Below are the definition of the boolean operators $\neg$, $\wedge$, $\forall$, $\exists$, and so on, lifted to the predicate level.

```
pSEQ_DEF: % everywhere operator %
  |- !p. |== p = (!s. p s)

RES_qOR:
  |- !W P. (??i::W. P i) = (\s. ?i. W i /\ P i s)

RES_qAND:
  |- !W P. (!!i::W. P i) = (\s. !i. W i ==> P i s)

EQUAL_DEF:
  |- !p q. p EQUAL q = (\s. p s = q s)

pIMP_DEF:
  |- !p q. p IMP q = (\s. p s ==> q s)
```

```
pOR_DEF:
  |- !p q. p OR q = (\s. p s \/ q s)

pAND_DEF:
  |- !p q. p AND q = (\s. p s /\ q s)

pNOT_DEF:
  |- !p. NOT p = (\s. ~p s)

FF_DEF:
  |- FF = (\s. F)

TT_DEF:
  |- TT = (\s. T)
```

A.4 Variables, Actions and Expressions

Below are some definitions and results which were discussed in sections 6 and 9.7.

```
Pj_DEF: % projection on functions %
  |- !V A x. (V Pj A)x = (A x => V x | Nov)

CONF_DEF: % predicate confinement %
  |- !A p.
     A CONF p
     =
     (!s t. (s Pj A = t Pj A) ==> (p s = p t))

HOA_DEF: % Hoare triple %
  |- !p A q.
     HOA(p,A,q) = (!s t. p s /\ A s t ==> q t)

a_Pj_DEF: % projection of actions %
  |- !a A.
     a_Pj a A = (\s t. a (s Pj A) (t Pj A))

rINTER
  |- !a b. a rINTER b = (\s t. a s t /\ b s t)

ALWAYS_ENABLED: % always enabled action %
  |-  !a. ALWAYS_ENABLED a = (!s. ?t. a s t)

SKIP_DEF
  |- SKIP = (\s t. s = t)

Update_DEF
  |- !v E. Update (v,E) = (\s t. t v = E s)

Assign_DEF
  |- !lv le.
      Assign(lv,le) = (\s t.
      (LENGTH lv = LENGTH le) /\ (LENGTH le) > 0)
      rINTER
      ((FOLDR $rINTER(\s t. T)(MAP Update(ZIP(lv,le))))
       rINTER
       (SKIP a_Pj (\x. ~IS_EL x lv))
      )

Guard_Action_DEF
  |- !g a1 a2.
      Guard_Action g a1 a2 =
      (\s t. (g s ==> a1 s t) /\ (~g s ==> a2 s t))
```

```
COND_EXPR_DEF
  |- !P E1 E2. COND_EXPR P E1 E2
        = (\s. (P s => E1 s | E2 s))

VAR_EXPR_DEF  |- !v. VAR_EXPR v = (\s. s v)

LT_DEF  |- !E1 E2. E1 LT E2 = (\s. (E1 s) < (E2 s))

IG_BY_DEF: % ignored variables %
  |- !V A.
     V IG_BY A
     =
     (!s t. A s t ==> (s Pj V = t Pj V))

INVI_DEF: % invisible variables %
  |- !V A.
      V INVI A =
      (!s t s' t'.
        (s Pj (NOT V) = s' Pj (NOT V)) /\
        (t Pj (NOT V) = t' Pj (NOT V)) /\
        (s' Pj V = t' Pj V) /\
        A s t ==>
        A s' t')

Exp_INVI_DEF
  |- !V E. V Exp_INVI E =
        (!s s'. (s Pj (NOT V) = s' Pj (NOT V))
                   ==> (E s = E s'))

ALL_ORD_GUARD_SAT_RC
  |- !Ord.
        (NOT R) Exp_INVI E1 /\ (NOT R) Exp_INVI E2 ==>
        (NOT R) Exp_INVI (\s. Ord(E1 s)(E2 s))

Assign_SAT_RC
  |- (!v. IS_EL v lv ==> R v) /\
       (!E. IS_EL E le ==> (NOT R) Exp_INVI E) ==>
       (NOT R) INVI (Assign(lv,le))

Guard_Action_SAT_RC
  |- (NOT R) Exp_INVI G /\
     (NOT R) INVI a1 /\ (NOT R) INVI a2 ==>
       (NOT R) INVI (Guard_Action G a1 a2)
```

A.5 Core UNITY

Below are the definition of the predicate Unity, defining the well-formedness of a UNITY program, and the definitions of all basic UNITY operators.

```
UNLESS:
  |- !Pr p q.
     UNLESS Pr p q =
     (!A :: PROG Pr. HOA(p AND (NOT q),A,p OR q))

ENSURES:
  |- !Pr p q.
     ENSURES Pr p q =
     UNITY Pr /\
     UNLESS Pr p q /\
        (?A :: PROG Pr. HOA(p AND (NOT q),A,q))

LEADSTO:
  |- !Pr. LEADSTO Pr = TDC(ENSURES Pr)

STABLE:
  |- !Pr p. STABLE Pr p = UNLESS Pr p FF

Inv: % invariant %
  |- !Pr J.
```

```
     Inv Pr J =
     |==((INIT Pr) IMP J) /\ UNLESS Pr J FF

PROG: % the action set of a program %
  |- !P In R W. PROG(P,In,R,W) = P

WRITE: % the write variables set of a program %
  |- !P In R W. WRITE(P,In,R,W) = W

READ:  % the read variables set of a program %
  |- !P In R W. READ(P,In,R,W) = R
```

```
INIT: % the initial condition of a program %
  |- !P In R W. INIT(P,In,R,W) = In

UNITY: % define a UNITY program %
  |- !P In R W.
     UNITY(P,In,R,W) =
     (!A :: P. ALWAYS_ENABLED A) /\
     (!A :: P. (NOT W) IG_BY A) /\
     (!x. W x ==> R x) /\
     (!A :: P. (NOT R) INVI A)
```

A.6 Convergence

The convergence operator is defined as follows in HOL:

```
CON:
  |- !Pr J p q.
     CON Pr J p q =
     (WRITE Pr) CONF q /\
     (?q'. REACH Pr J p(q' AND q) /\ STABLE Pr(q' AND (q AND J)))
```

Below is a list of some its basic properties, which were also depicted in Figure 11.

```
CON_ENSURES_LIFT:
  |- !Pr J p q.
     (WRITE Pr) CONF p /\ (WRITE Pr) CONF q /\
      STABLE Pr J /\ STABLE Pr (q AND J) /\
      ENSURES Pr (p AND J) q
      ==>
      CON Pr J p q);

CON_REFL:
  |- !Pr J p.
     UNITY Pr /\ (WRITE Pr) CONF p /\
     STABLE Pr J /\ STABLE Pr(p AND J)
     ==>
     CON Pr J p p

CON_SUBST:
  |- !Pr J p q r s.
     (WRITE Pr) CONF r /\ (WRITE Pr) CONF s /\
     |==((r AND J) IMP p) /\
     |==((q AND J) IMP s) /\
     CON Pr J p q
     ==>
     CON Pr J r s
```

```
CON_WF_INDUCT: % bounded progress induction
                 principle for convergence %
  |- ADMIT_WF_INDUCTION LESS /\
     CON Pr J q q /\
     (!m. CON Pr J
              (p AND (\s. M s = m))
              ((p AND (\s. LESS(M s)m)) OR q))
     ==>
     CON Pr J p q

CON_BY_sWF_i: % the round decomposition
                principle %
  |- sWF A U /\ ~(A = {}) /\ FINITE A /\
     STABLE Pr J /\
     (!y::\y. y IN A.
          CON Pr
              (J AND
               (!! x :: \x. x IN A /\ U x y. Q x))
              TT
              (Q y))
     ==>
     CON Pr J TT(!! y :: \y. y IN A. Q y)
```

A.7 Parallel Composition

Below is the definition of UNITY parallel composition together with some other operators.

```
PAR: % parallel composition of programs %
  |- !Pr Qr.
     Pr PAR Qr =
       ( PROG Pr) OR (PROG Qr),
         (INIT Pr) AND (INIT Qr),
         (READ Pr) OR (READ Qr),
         (WRITE Pr) OR (WRITE Qr)
```

```
UNLESS_PAR_i:
  |- UNLESS Pr p q /\ UNLESS Qr p q
     =
     UNLESS(Pr PAR Qr)p q

STABLE_PAR_i:
  |- STABLE Pr p /\ STABLE Qr p
     =
     STABLE(Pr PAR Qr)p

Inv_PAR:
  |- !J Pr Qr.
     Inv Pr J /\ Inv Qr J ==> Inv(Pr PAR Qr)J

ENSURES_PAR:
  |- !Pr :: UNITY. !p q Qr.
     UNLESS Pr p q /\ ENSURES Qr p q
     ==>
     ENSURES(Pr PAR Qr)p q

CON_TRANSPARANT:
  % transparency law for convergence %
  |- !Pr Qr J p q.
     UNITY Qr /\ Pr <w> Qr /\
     STABLE Qr J /\ CON Pr J p q
     ==>
     CON(Pr PAR Qr)J p q
```

A.8 The sorting program and related definitions

```
Sorted_DEF
  |- !P A vOrd pOrd.
     Sorted P A vOrd pOrd =
     (!i j.
       i IN P /\ j IN P /\ pOrd i j
       ==> vOrd(A i)(A j)
     )

Wrong_Pairs_DEF
  |- !P A vOrd pOrd.
      Wrong_Pairs P A vOrd pOrd =
       {(i,j)
        | i IN P /\ j IN P /\
          pOrd i j /\ ~vOrd(A i)(A j)}

Wrong_Pair_DEF
  |- !P A i j vOrd pOrd.
     Wrong_Pair P A i j vOrd pOrd =
     i IN P /\ j IN P /\
     pOrd i j /\ ~vOrd(A i)(A j)


Nr_Of_Wrong_Pairs_DEF
  |- !P A vOrd pOrd.
    Nr_Of_Wrong_Pairs P A vOrd pOrd
     = CARD(Wrong_Pairs P A vOrd pOrd)

Nr_Of_Wrong_Pairs_GREATER_0_IMP_EXISTS_WP
  |- !P A vOrd pOrd s.
     FINITE P /\ SORT_ORDER vOrd P A ==>
     (Nr_Of_Wrong_Pairs P(s o A)vOrd pOrd) > 0
      ==>
     (?p.
       CHF(POSSIBLE_CONNECTIONS P) p /\
       Wrong_Pair P (s o A)(FST p)(SND p)vOrd pOrd)


Two_Elts_Swapped_DEF
  |- !P C A A' vOrd pOrd.
      Two_Elts_Swapped P C A A' vOrd pOrd =
      (?i j.
        i IN P /\
        j IN P /\
        pOrd i j /\
        Connected(i,j)C /\
        (!k. k IN P /\ ~(k = i) /\ ~(k = j)
            ==> (A k = A' k)
        ) /\ (A i = A' j) /\
        (A j = A' i) /\
        ~vOrd(A' i)(A' j))

Sufficient_Connections_In_Network_For_Sorting
|- !pOrd NW.
   Sufficient_Connections_In_Network_For_Sorting pOrd NW
    =
   (p2s pOrd) SUBSET
   (Transitive_Refl_Closure
    (p2s pOrd) INTER (Symmetric_Closure(Conns NW))
    (Procs NW))

EXISTS_AT_LEAST_ONE_CONNECTION_THAT_CAN_BE_A_WP
|- !P C A pOrd.
   Network(P,C,A) /\
   Sufficient_Connections_In_Network_For_Sorting pOrd(P,C,A)
    ==>
   (?i j. pOrd i j /\ Connected(i,j)C /\ i IN P /\ j IN P)

Nr_of_Wrong_Pairs_GREATER_0_IMP_EXISTS_CONNECTED_WP
|- !P C A vOrd pOrd s.
   Sufficient_Connections_In_Network_For_Sorting pOrd(P,C,A)
   /\ INDEX_ORDER pOrd P /\
   Network(P,C,A) /\
   (Nr_Of_Wrong_Pairs P(s o A)vOrd pOrd) > 0 /\
   FINITE P /\
   SORT_ORDER vOrd P A ==>
   (?e :: CHF{(i,j) | i IN P /\ j IN P /\ pOrd i j
          /\ Connected(i,j)C}.
     Wrong_Pair P(s o A)(FST e)(SND e)vOrd pOrd)

Sort_PROG
  |- !P C A A' vOrd pOrd.
      Sort(P,C,A)A' vOrd pOrd =
      CHF
      {Assign2(A i,A j)(Min vOrd(A i)(A j),Max vOrd(A i)(A j)
       | i IN P /\ j IN P /\ pOrd i j /\ Connected(i,j)C},
      (\s. VALUES_EQ A A' s),CHF{A i | i IN P},CHF{A i | i IN P}


MAIN_THEOREM_FOR_BOUNDED_PROGRESS
  |- !s t A A'' vOrd pOrd m.
      SORT_ORDER vOrd P A /\
      INDEX_ORDER pOrd P /\
      FINITE P /\
      Values_EQ (s o A)A'' /\
      (Nr_Of_Wrong_Pairs P(s o A)vOrd pOrd = m) /\
      Two_Elts_Swapped P C(t o A)A'' vOrd pOrd ==>
      (Nr_Of_Wrong_Pairs P(t o A)vOrd pOrd) < m

ENSURES_COND_FOR_SORTING_PROGRAM
  |- !P C A pOrd vOrd A.
      SORT_ORDER vOrd P A /\
      INDEX_ORDER pOrd P /\
 i IN P /\
 j IN P /\
 pOrd i j /\
 Connected(i,j)C /\
 m > 0 /\
 FINITE P ==>
 ENSURES
 (Sort(P,C,A)A' vOrd pOrd)
 (((\s. Nr_Of_Wrong_Pairs P(s o A)vOrd pOrd = m) AND
 (WRONG_PAIR P A i j vOrd pOrd)) AND
(PERM P A A'))
 (\s. (Nr_Of_Wrong_Pairs P(s o A)vOrd pOrd) < m)
```

```
SORTING_PROG_SPECIFICATION
|- !P C A pOrd vOrd A.
    Network(P,C,A) /\
    Sufficient_Connections_In_Network_For_Sorting pOrd(P,C,A)
    /\ SORT_ORDER vOrd P A /\
    INDEX_ORDER pOrd P ==>
    (!A'.
      CON
      (Sort(P,C,A)A' vOrd pOrd)
      (PERM P A A')
      (VALUES_EQ A A')
      (SORTED P A vOrd pOrd))
```

References

[And92] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[BBL93] J.P. Bowen, P Breuer, and Kevin Lano. A compendium of formal techniques for software maintenance. *IEE/BCS Software Engineering Journal*, 8(5):253–262, September 1993.

[BGM90] J. Burns, M. Gouda, and R. Miller. Stabilization and pseudo-stabilization. Technical report, University of Texas, TR-90-13, May 1990.

[BH95a] J.P. Bowen and M Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] J.P. Bowen and M Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[Bou92] R. Boulton. The hol arith library. Technical report, Computer Laboratory University of Cambridge, July 1992.

[Bou93] Richard John Boulton. *Efficiency in a Fully-Expansive Theorem Prover.* PhD thesis, University of Cambridge, Decembre 1993.

[Bou94] R.J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, 1994.

[Bou95] Richard J. Boulton. Combining decision procedures in the HOL system. In E. T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 75–89, Aspen Grove, Utah, USA, September 1995. Lecture Notes in Computer Science volume 971, Springer-Verlag.

[Bow93] J.P. Bowen. Formal methods in safety-critical standards. In *Proc. 1993 Software Engineering Standards Symposium (SESS'93)*, pages 168–177, Brighton, UK, 30 August - 3 September 1993. IEEE Computer Society Press.

[BS92] J.P. Bowen and V. Stavridou. Formal methods and software safety. In H.H. Frey, editor, *Safety of computer control systems 1992 (SAFECOMP '92)*, pages 93–98, Zürich, Switzerland, October 1992. Proc. IFAC Symposium, Pergamon Press.

[BS93a] J.P. Bowen and V Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 183–195. Springer-Verlag, LNCS 670, 1993.

[BS93b] J.P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEEE Software Engineering Journal*, 8(4):189–209, July 1993.

[Bus94] H. Busch. First-order automation for higher-order-logic theorem proving. In T.F. Melham and J. Camilleri, editors, *Lecture Notes in Computer Science 859: Higher Order Theorem Proving and Its Application*, pages 97–122. Springer-Verlag, 1994.

[But93] R.W. Butler. Formal methods for life-critical software. In *AIAA Computing in Aerospace 9 Conference*, pages 319–329, San Diego, October 19-21 1993.

[BvW90] R.J.R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, (2):247–272, 1990.

[BW88] R. Bird and Philip Wadler. *Introduction to functional programming.* Prentice Hall, 1988.

[CG92] D. Craigen and S. Gerhart. An international survey of industrial applications of formal methods. In *Z User Workshop, London 1992*, pages 1–5. Springer-Verlag, 1992.

[CGR93] D. Craigen, S. Gerhart, and T Ralston. Formal methods reality check: Industrial usage. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 250–267. Springer-Verlag, LNCS 670, 1993.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[Coo72] D.C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence* γ, chapter 5, pages 91–99. Edinburgh University Press, 1972.

[DDH72] J. Dahl, O, E.W. Dijkstra, and C.A.R. Hoare. *Structured programming.* Academic Press, Orlando, 1972.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, January 1994.

[GM93] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[Kem90] R.A.. Kemmerer. Integrating formal methods into the development process. *IEEE Software*, pages 37–50, September 1990.

[KKS91] R. Kumar, T. Kropf, and K. Schneider. Integrating a first order automatic prover in the hol environment. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications.* IEEE Computer Society Press, August 1991.

[Lån94] T. Långbacka. A hol formalization of the temporal logic of actions. In T.F. Melham and J. Camilleri, editors, *Lecture Notes in Computer Science 859: Higher Order Theorem Proving and Its Application*, pages 332–345. Springer-Verlag, 1994.

[Len93] P.J.A. Lentfert. *Distributed Hierarchical Algorithms.* PhD thesis, Utrecht University, April 1993.

[LT93] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, pages 18–41, July 1993.

[Mel90] T.F. Melham. The HOL sets library. *http://128.187.2.182/lal/holdoc/library.html*, 1990.

[Neu95] Peter G. Neumann. *Computer Related Risks.* Addison-Wesley Publishing Compagny, 1995.

[Nic91] J.E. Nicholls. Domains of application for formal methods. In *Z User Workshop, York 1991*, pages 145–156. Springer-Verlag, 1991.

[Pau87] Lawrence C. Paulson. *Logic and Computation:Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[Pra95] Wishnu Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* PhD thesis, Utrecht University, October 1995.

[Rus94] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[RvH93] J. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[SB94] Tom Schubert and John Biggs. A tree-based, graphical interface for large proof development. In *1994 International Workshop on the HOL Theorem Proving System and its Applications*, 1994.

[SKR91] K. Scneider, T. Kropf, and Kumar R. Integrating a first-order automatic prover in the hol environment. In M Archer, Joyce J.J., Levitt K.N., and Windley P.J., editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications.* IEEE Computer Society Press, 1991.

[Sym95] Donald Syme. A new interface for HOL - ideas, issues and implementation. In *1995 Conference on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, 1995.

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. Technical Report 1684, Institut National de Recherche en Informatique et en Automatique., 1992.

[Thé93] Laurent Théry. A proof development system for the HOL theorem prover. In *International Workshop on Higher Order Logic and its applications*, Vancouver, 1993.

[Tho94a] Muffy. Thomas. A proof of incorrectness using the lp theorem prover: the editing problem in the Therac-25. *High Integrity Systems Journal*, 1(1):35–48, 1994.

[Tho94b] Muffy. Thomas. The story of the Therac-25 in lotos. *High Integrity Systems Journal*, 1(1):3–15, 1994.

[WW93] D Weber-Wulff. Selling formal methods to industry. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 671–678. Springer-Verlag, LNCS 670, 1993.

Index

$[p]$, 17
Action, 17
$\sqcap$, 18
$\not\leftarrow$, 21
$\Box$, 24
$\not\twoheadrightarrow$, 22
$\not\twoheadrightarrow_e$, 65
$\|$, 26
State, 15
Var, 15
unless$_V$, 28
Val, 15
$\aleph$, 15
S^{c}, 17
ensures , 23
$\mapsto$, 26
free.p, 17
$\restriction$, 15
$\rightarrowtail$, 28
$\circlearrowright$, 24
unless , 23
$p \in$ Pred.V, 17
*val, 34
*var, 34
define_type, 34

action, 17
    always enabled, 18
    assignment, 17, 18
    atomic, 19
    guarded, 17
    multiple assignment, 18
    skip, 17
always enabled action, 18
anti-quotations, 9
assignment action, 17
axiom
    BOOL_CASES_AX, 10
    ETA_AX, 10
    IMP_ANTISYM_AX, 10
    INFINITY_AX, 10
    SELECT_AX, 10
    extending HOL by adding an, 11

backward proof, 12

CHOL, 76
closure
    symmetric, 44
    transitive reflexive, 44
closure, transitive reflexive, 44
compositionality, 6, 27
confinement
    predicate, 17
convergence, 6, 31
    bounded progress property, 33
    conjunction property, 33
    definition, 31
    disjunction property, 33
    reflexivity property, 33
    transitivity property, 33

decision procedure, 14, 77
    first order predicate logic, 14
    Presburger arithmetic
        ARITH_CONV, 14
    propositional logic, 14

embedding
    deep, 34
    shallow, 34
extension
    conservative, 11
    definitional, 11

formal methods
    three levels of usage, 4
    what is meant by, 4
formal proof, 9
formal specification
    advantages, 5
forward proof, 12
FSA algorithm, 6
fully-expansive theorem prover, 77
function
    bijective, 45
    equality, 45
    injective, 45

    surjective, 45

goal, 12
graph
    directed, 44
    undirected, 45
guarded action, 17

higher order logic, 7
hoare triple, 17
    HOL definition, 12
HOL
    's 8 primitives inference rules, 10
    's five axioms, 10
    compound types, 9
    formula, 8
    logical terms, 8
    primitive types, 9
    theorem, 9
    type constructors, 9
    type-variables, 9
    user interfaces, 76
HOL2000, 75

ignored-by
    $\nleftarrow$ (variables by actions), 21
inference rule
    derived, 9, 10
        `MATCH_MP`, 10
        `REWRITE_RULE`, 10
    primitive, 9
        `ABS`, 10
        `ASSUME`, 10
        `BETA_CONV`, 10
        `DISCH`, 10
        `INST_TYPE`, 10
        `MP`, 10
        `REFL`, 10
        `SUBST`, 10
invariant
    definition, 24
invisible variables, 22
invisible-to
    $\not\twoheadrightarrow$ (variables to actions), 22
    $\not\twoheadrightarrow_e$ (variables to expressions), 65

justification function, 13

LCF (Logic of Computable Functions), 7
library
    `arith`, 14
    `faust`, 14
    `taut`, 14

meta-language types, 8
model checking, 78
modularity principle, 27
multiple assignment action, 18

network of processes
    sufficient connections for sorting, 50

object language types, 8
overloading of symbols, 16

parallel composition
    definition, 26
    modelling of, 26
pencil-and-paper proofs
    the advantages, 42
permutation
    definition, 51
predicate confinement
    definition, 17
    HOL definition, 35
program state, 15
program union, 26
progress property, 22
    ensures
        compositionality property, 27
        conjunction property, 24
        definition, 23
        post-weakening, 24
        PSP property, 24
    leads-to($\mapsto$)
        definition, 26
    reach ($\rightarrowtail$)
        bounded progress property, 29
        definition, 28
        disjunction property, 29
        PSP property, 29
projection
    of a function, 15
    of an action, 18

proof
  formal, 9
  forward, 12
  goal-directed,top-down,backward, 7, 12
proof tree, 12
pseudo-stabilisation, *see* convergence

relation
  anti-reflexive, 44
  anti-symmetric, 44
  binary, 43
  partial order, 44
  reflexive, 44
  symmetric, 44
  symmetric closure, 44
  total order, 44
  transitive, 44
  transitive reflexive closure, 44
restriction, *see* projection
round decomposition, 32
  principle, 32

safety critical systems, 5
safety property, 22
  unless
    compositionality property, 27
    conjunction property, 25
    definition, 23
    disjunction property, 25
    post-weakening, 25
safety-critical processes, 4
self-stabilisation, 6, 31
Singh law, 29
skip action, 17
solution strategy, 52
specification
  decomposition, 41
  formal, 40
  language, 41
  refinement, 41
stable
  predicate
    compositionality property of, 27
    conjunction property of, 25
    definition, 24
    disjunction property of, 25
  properties, 24
state
  expression, 17
  of a program, 15
  predicate, 16
statement, *see* action
sub-goal package, 14
sub-goals, 12
synchronisation operator, 18
  rINTER, 18

tactic, 12
  `ACCEPT_TAC`, 12
  `CONJ_TAC`, 12
  `MATCH_MP_TAC`, 12
  `RES_TAC`, 12
  combinators, *see* tactical
  higher order, *see* tactical
tactical, 13
  `ORELSE`, 13
  `REPEAT`, 13
  `THEN`, 13
Therac-25, 4, 75
TkHOL, 76
transparency Law, 30

UNITY, 15
  's fairness condition, 19
  program development methodology, 41
  specifications, 22
UNITY program
  execution of a, 19
  syntax of a, 20
  well-formedness of a, 21
    HOL definition, 38
universe of
  action
    Action, 17
    HOL definition, 36
  program states
    State, 15
    HOL definition, 35
  program variables
    Var, 15
  state predicates
    Pred, 16

HOL definition, 35
values
Val, 15

well-formedness of a UNITY program, 21
WrongPair of processes, 49

xhol, 76